

# Dynamic Data, Lists, and Class Templates

# 12

Something deeply hidden had to be behind things.

Albert Einstein

*autobiographical handwritten note, The Einstein Letter That Started It All  
NY Times Magazine, August 2, 1964, Ralph E. Lapp*

Although `tvector` variables can be resized and increase (or decrease) in capacity, excess storage is often allocated when vectors are used. Since vectors typically double in size when grown, memory will be wasted unless all vector cells are used. For example, consider a program that counts how many times each of the 3,124 unique words in the file `melville.txt` (*Bartleby, the Scrivener*) occurs by storing the words in a vector using `push_back`. The vector grows in size from 0 to 2, to 4, 8, 16, ... 4,096 elements. Since the automatic resizing operation throws out the old vector after copying elements into a new vector, a total of  $2 + 4 + \dots + 2048 = 4,094$  elements are thrown out while  $4096 - 3124 = 972$  elements in the final vector are unused. Although the `tvector` class takes the necessary step to reclaim the storage thrown away, some applications require more precise memory allocation. We've also studied an example of a sparse polynomial class (see Programs 10.18 and 10.19) that was more efficiently implemented using a `CList` collection of terms than a `tvector` collection. In this chapter we'll study a data structure called a **linked list** which provides an alternative to using vectors. We'll also study how **pointers**, which are used in implementing linked lists and trees, expand the kinds of programs we can write. Pointers are essential in working with large object-oriented programs in C++ and in exploiting inheritance which we'll cover in Chapter 13. However, once we use pointers, we have to be careful in designing classes to avoid problems we haven't faced before.

## 12.1 Pointers as Indirect References

### 12.1.1 What is a Pointer?

We'll cover three basic uses of pointers in this chapter.

1. Pointers are indirect references that permit resources to be shared among different objects. For example, several random walkers could share an object that records all their positions, or shows the positions graphically. Without pointers it's not possible to share an object and to change which object is shared among all the walkers.
2. Pointers let code allocate memory **dynamically**, on an as-needed basis during program execution rather than when the program is compiled. The programmer

controls the lifetime of dynamically allocated memory unlike the **statically** allocated memory we've used so far. Here static is used as the opposite of dynamic, not to mean allocating static variables as discussed in Section 10.4.3. The variables we've used so far have a lifetime determined by the variable's scope.<sup>1</sup>

3. Pointers are the basis for implementing linked data structures which are used in many applications. We'll see how linked lists are the basis for the implementation of the class `CList` and how they are used to implement a set class similar to `StringSet`.

At a basic level, a pointer stores an **address** in computer memory. More abstractly, a pointer refers to something indirectly. If you look up *pointer* in the index of this book, you'll see a reference, or "pointer," to this page. Forwarding addresses also serve as indirect references. Suppose someone named Dave Reed lives at 104 Oak Street. If Dave moves, he'll leave a forwarding address with the post office. If he moves to 351 Coot Lane, then mail addressed to him at 104 Oak Street will be delivered, with some delay, using the forwarding address. The forwarding address is a pointer, or indirect reference, to Dave's new address.

Program 12.1, *pointerdemo.cpp*, shows how pointers are defined and **dereferenced**. A pointer variable is defined when an asterisk `*` appears between a type/class name and a variable name. A pointer is an address, but it's an address of a specific type of object, such as `Date`, `int`, or any other built-in or class type. Just as `int x;` defines variable `x` with no value, a pointer has no value unless one is assigned.

#### Syntax: Pointers

```
Date * d; // points to garbage
Date * d = new Date();
Date * d2 = d;
Date next = *d + 1;
int month = d->Month();
```

We'll allow pointers to point to (or reference) objects in two ways: allocating an object using **new** or sharing objects between pointers. The **new** operator returns a pointer to an object created on the **heap**; we'll say more about this later. We can also assign one pointer value to another as shown with `d` and `d2`. To access the object pointed to by a pointer `p`, the expression `*p` is used, where `*` is the **dereference** operator. To select a member function in an object pointed to by `p` we'll use the **selector** operator `->` and write `p->Function()`. The selector operator is shorthand for writing `(*p).Function()`, where dereferencing `p` yields an object on which the method `Function` is invoked.

---

#### Program 12.1 pointerdemo.cpp

---

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

---

<sup>1</sup>For example, the lifetime of a variable declared locally in a function is the duration of the function. See Section 10.4 for details.

```

#include "dice.h"

// basic pointer demo

int main()
{
    Date today;
    Date * nextDay = new Date(today+1);
    Date * prevDay = new Date(today-1);

    cout << "today\t\t tomorrow\t\tyesterday" << endl;
    cout << today << "\t" << nextDay << "\t" << prevDay << endl;
    cout << today << "\t" << *nextDay << "\t" << *prevDay << endl;

    nextDay = prevDay;
    cout << today << "\t" << *nextDay << "\t" << *prevDay << endl;
    *prevDay += 2;
    cout << today << "\t" << *nextDay << "\t" << *prevDay << endl;
    cout << today << "\t" << nextDay << "\t" << prevDay << endl;

    cout << endl << "k\sides\troll\tcount" << endl <<endl;
    const int DICE_COUNT = 6;
    tvector<Dice *> dice(DICE_COUNT);
    int k;
    for(k=0; k < DICE_COUNT; k++)
    {
        dice[k] = new Dice(2*k+1);
    }
    for(k=0; k < DICE_COUNT; k++)
    {
        cout << k << "\t" << dice[k]->NumSides() << "\t"
            << dice[k]->Roll() << "\t";
        cout << dice[k]->NumRolls() << endl;
    }
    return 0;
}

```

---

pointerdemo.cpp

Memory addresses in C++ are typically shown using the base 16, or **hexadecimal**, number system, where the letter *a* corresponds to 10, *b* to 11, and so forth, with *f* corresponding to 15. Don't worry about trying to understand hexadecimal notation; you can think of addresses as having values like "101 Main Street." The important relationship is that the value of a pointer is an address. In the output from Program 12.1, the printed values of the pointers `nextDay` and `prevDay` are the addresses of what each points to in memory. When the pointers are dereferenced, for example, in the expression `*nextDay`, the object being pointed to, a `Date`, is printed.

I ran Program 12.1 on May 18, 1999. The first line of output shows that `nextDay` and `prevDay` point to different objects since the addresses printed are different. The last line of `Date` output shows that these pointers refer to the same object since the addresses are the same. Since the two pointers refer to the same object, when that object is incremented by two in the statement `*prevDay += 2`, what happens to the value of `*nextDay`? Since `*nextDay` is "the object pointed to by `nextDay`"<sup>2</sup>, and this

---

<sup>2</sup>I pronounce `*nextDay` as "star nextDay." Sometimes I say "the object pointed to by `nextDay`" to be precise.

object is the same object as `*prevDay`, the statement `*prevDay += 2` affects what `nextDay` points to as well.

```

O U T P U T

prompt> pointerdemo
today          tomorrow          yesterday
May 18 1999    0x00142a10        0x00142a20
May 18 1999    May 19 1999       May 17 1999
May 18 1999    May 17 1999       May 17 1999
May 18 1999    May 19 1999       May 19 1999
May 18 1999    0x00142a20        0x00142a20

k      sides  roll  count
0      1      1     1
1      3      2     1
2      5      3     1
3      7      4     1
4      9      3     1
5     11      5     1
    
```

The second part of the program creates a vector of `Dice` pointers and rolls each of the `Dice` objects once. Recall that it's not possible to create a `tvector<Dice>` variable because there is no default `Dice` constructor. However, a vector of `Dice` pointers can be created as shown in Figure 12.1.

When the vector is defined, the six pointers do not have specific values, they point at "garbage." The word "garbage" means the value of a pointer may be something like 6 or it may be something like `0xffde2000`; we don't know if the value is a valid memory location. We create a separate `Dice` object on the heap for each vector pointer to

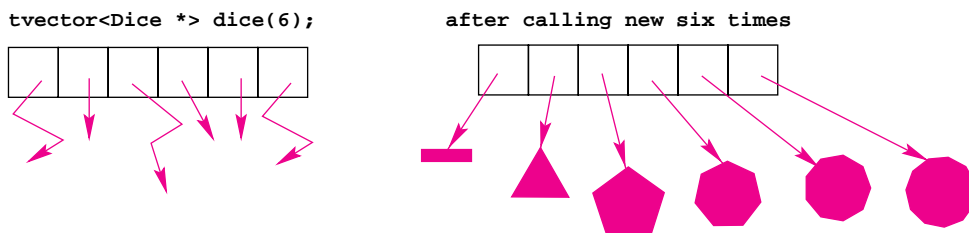


Figure 12.1 A vector of pointers to `Dice` objects.

reference, each `Dice` object with a different number of sides as shown in the code and the output. The selector operator `->` accesses the member functions of each pointed-to `Dice` object. I pronounce `d->NumRolls()` as “d arrow NumRolls”, but sometimes I say “the NumRolls method of the object pointed to by d.” The latter pronunciation makes the pointer/pointed-to difference very clear. A few programmers prefer to write `(*d).NumRolls()`. The dot operator `.'` has higher precedence than the dereference operator `*`, so parentheses are needed in the expression `(*d).Roll()`. Otherwise, the expression `*d.Roll()` results in an attempt to dereference the `Roll()` function of `d`. This would fail for two reasons:

- `d` is not a class object, so a dot can't follow it.
- `Roll()` is not a pointer, so it can't be dereferenced (assuming that `d.Roll()` made syntactic sense).

Most programmers prefer `->`, the selector operator which is typed using the minus sign followed by the greater-than sign. It's easier to read and type `p->foo()` than `(*p).foo()`.

### 12.1.2 Heap Objects

The variables `today`, `nextDay`, and `prevDay` are defined in the function `main` of Program 12.1. The memory for these variables is associated with the function. In general, variables defined in a function are constructed when the function is called and cease to exist when the function exits. These variables are called **automatic** variables since the memory for them is automatically allocated when the function is called and de-allocated when the function exits. Sometimes the term **stack** variable is used and memory is said to be “allocated on the runtime stack” for variables defined in a function.

In contrast, memory allocated by calling **new** is obtained dynamically when the `new` statement executes, not automatically. The memory initialized by `new` is allocated from the **heap**, sometimes called the **freestore**. Objects constructed on the heap last until the program specifically de-allocates them (using **delete**, which we'll discuss in Section 12.1.7) unlike automatic variables which are de-allocated automatically when they go out of scope. If the class/type allocated by `new` uses a constructor, then

#### Syntax: The `new` operator

```
Thing * t = new Thing;
Thing * t = new Thing();
Thing * t = new Thing(parameters);
```

arguments must be provided if the constructor requires them as shown in the vector of `Dice` pointers of Program 12.1. A constructor with no parameters does not require parentheses so that `Date * d = new Date;` creates an object representing `today` that's pointed to by `d`. Parentheses may be used, the statement `Date * d = new Date();` creates the same kind object. I'll use parentheses even when no parameters are passed to the constructor.

In this book I'll use pointers only to point to objects on the heap. In C++ it's possible for objects to point to memory on the stack as well. Invariably this leads to problems

because memory on the stack “goes away” when a scope ends. A pointer to stack memory that is out-of-scope will eventually cause problems if the pointer is dereferenced. To help you read programs written by others, I’ll show how the **address-of** operator & is used to get the address of stack variables, but it’s a good idea to stay away from the address-of operator until you’re a reasonably accomplished programmer.

```
int main()
{
    Date * d = new Date();           // d points to today
    Date * d2 = new Date(*d+1);     // d2 points to tomorrow
    Date * d3;                       // d3 points to garbage
    if (*d < *d2)
    {
        Date yday(*d-1);           // yesterday, all my troubles ...
        d3 = &yday;                // d3 points to yesterday
        cout << "yesterday " << *d3 << endl;
    }
    cout << *d3 << " " << *d << " " << *d2 << endl;
    return 0;
}
```

If I run this program on May 15, 1999, the output will be unpredictable:

O U T P U T

```
yesterday May 14 1999
????? May 15 1999   May 16 1999
```

The code is problematic: `d3` points to an object that doesn’t exist. The address-of operator & applied to `yday` returns the address of `yday`. This works as intended in the body of the `if` statement, but the variable `yday` doesn’t exist after the body of the `if` statement executes. This means that `d3` points to a nonexistent object, what’s printed depends on a number of unknown factors including how the compiler works and how the operating system behaves. The program may produce what’s expected the first time it runs, but not the second.

**Program Tip 12.1: Memory referenced by a pointer should be allocated from the heap.** Using the address-of operator to obtain the address of memory allocated on the stack will eventually lead to problems if that memory goes out of scope. Tracking down this kind of error is difficult because the error often manifests itself differently on different runs of the program and sometimes occurs in code unrelated to where the address-of operator is used.

## Pause to Reflect



- 12.1** Write code that defines two `Dice` pointers, allocates one 8-sided `Dice` object using `new` that both point to, and then rolls the `Dice` twice, once with each pointer.
- 12.2** Write a code fragment that defines a vector `dicevec` of 30 pointers to `Dice` objects, initializes `dicevec[k]` to point to a  $(k+1)$ -sided die (so that `dicevec[0]` is a one-sided die and `dicevec[29]` is a 30-sided die), and then rolls the dice object pointed to by `dicevec[k]`  $k$  times.
- 12.3** Write a code fragment that creates a vector `datevec` of pointers to `Date` objects. There should be as many pointers as there are days in the month the code is executed, (e.g., if run in April there should be 30 pointers, if run in May there should be 31 pointers). Initialize `datevec[k]` to point to an object representing the  $(k + 1)^{st}$  day of the month, so `datevec[0]` is the first day of the month. Print each day by looping over all the vector elements.
- 12.4** Write a function that returns a pointer to a `Date` object that represents exactly one year from the date the function is executed.
- 12.5** Consider the following function `MakeDie` that returns a pointer to a dice object.

```
Dice * MakeDie(int n)
// post: return pointer to n-sided Dice object
{
    Dice nSided(n);
    return &nSided;
}
```

Explain why this function can cause problems in code. In particular, the code below may print 6, 4, or some unknown value.

```
Dice * cube = MakeDie(6);
Dice * tetra = MakeDie(4);
cout << cube->NumSides() << endl;
```

When compiled under Linux/g++, the code generates a warning “*address of local variable ‘nSided’ returned*”.

- 12.6** In the worst case, selection sort makes  $O(N^2)$  comparisons and  $O(N)$  swaps and assignments to sort an  $N$ -element vector of strings. Insertion sort makes  $O(N^2)$  comparisons and  $O(N^2)$  object assignments. If vectors of pointers to strings are sorted rather than vectors of strings, insertion sort may speed up, while selection sort slows down. Explain these observations, think about how comparisons are made (how does the code change) and how objects are swapped/assigned. The change in execution time is less noticeable if `int` vectors are sorted (compared to `int *` vectors) and more noticeable if vectors of large `BigInt` objects are sorted (compared to `BigInt *` vectors).

**12.7** Suppose that the following definition is made:

```
tvector<tvector<int> *> v(10);
```

so that `v[0]` is a pointer to a vector of integers. The following code fragment makes `v[0]` point to a vector of 100 integers, all equal to 2.

```
v[0] = new tvector<int>(100,2);
```

Since `v[0]` points to a vector of 100 integers, how is an element of this 100-integer vector indexed? Write a loop to print all elements of the 100-element vector.

**12.8** What do you think happens if the `new` operator is called, but there is no memory on the heap? How could this happen in a program?

**12.9** If a vector of pointers to strings is sorted using operator `<` to compare the pointers, the output will be based on the addresses of the strings, (i.e., `a[0]` will be the string with the lowest numerical address in memory). Complete the function object `StrPtrCompare` to sort `vector<string *> a` so that the strings pointed to will be in alphabetical order.

```
struct StrPtrCompare
{
    int compare(string * lhs, string * rhs)
    {
        // fill in code here
    }
};
```

### 12.1.3 Sharing Objects

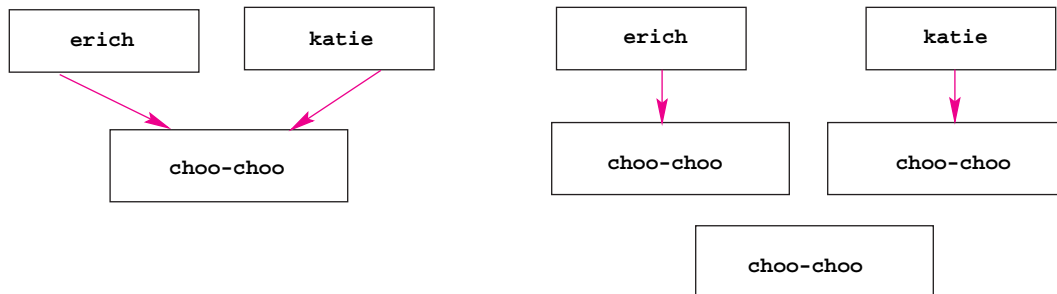
In this section we'll see how pointers make it possible to share an object. In all the classes we've used so far, the instance variables in one object are independent from the instance variables in any another object. Suppose that we want to create several random walk objects (see Programs 7.10 and 7.12), but keep in one vector a record of the positions visited by all the walkers. We'd like to share the vector among all the random walkers, but without pointers we cannot do this. We'll illustrate the problem with a more simple toy example before developing a solution for the posed problem with random walkers.

Program 12.2 shows what happens when two `Kid` objects try to share a `Toy`. The situation we'd like to have is illustrated in Figure 12.2. The figure and the program output show what actually happens.

---

#### Program 12.2 sharetoy.cpp

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



**Figure 12.2** Sharing without pointers. On the left what we want: objects `erich` and `katie` to share a toy. On the right what we have: three copies of a toy, no sharing.

```

using namespace std;

// references and pointers for sharing, a prelude

class Toy // kids play with toys
{
public:
    Toy(const string& name);
    void Play(); // prints a message
    void BecomeBroken(); // the toy becomes broken
private:
    string myName;
    bool myIsWorking;
};

class Kid
{
public:
    Kid(const string& name, Toy& toy);
    void Play(); // plays with own toy
private:
    string myName;
    Toy myToy;
};

Kid::Kid(const string& name, Toy& toy)
: myName(name), myIsWorking(true)
{ }

void Toy::Play()
// post: toy is played with, message printed
{
    if (myIsWorking)
    { cout << "this " << myName << " is so fun :-)" << endl;
    }
}

```

580

## Chapter 12 Dynamic Data, Lists, and Class Templates

```

        else
        { cout << "this " << myName << " is broken :-( " << endl;
        }
    }

void Toy::BecomeBroken()
// post: toy is broken
{
    myIsWorking = false;
    cout << endl << "oops, this " << myName << " just broke" << endl << endl;
}

Kid::Kid(const string& name, Toy& toy)
    : myName(name), myToy(toy)
{ }

void Kid::Play()
// post: kid plays and talks about it
{
    cout << "My name is " << myName << ", ";
    myToy.Play();
}

int main()
{
    Toy plaything("choo-choo train");
    Kid erich("erich", plaything);
    Kid katie("katie", plaything);

    erich.Play(); katie.Play();
    plaything.BecomeBroken(); // the toy is now broken
    erich.Play(); katie.Play();
    return 0;
}

```

sharetoy.cpp

Although the Toy object `plaything` is broken in `main`, the kids continue to enjoy a working toy. The problem is that the instance variable `myToy` in each kid is a **copy** of the toy defined in `main`. When we assign one variable to another, we don't expect the variables to share anything. In other words, we expect the output of the following statements to be "hello world," not "hello hello."

```

string a = "world";
string b = a;
a = "hello ";
cout << a << b << endl;

```

## O U T P U T

```

prompt> sharetoy
My name is erich, this choo-choo train is so fun :-)
My name is katie, this choo-choo train is so fun :-)

oops, this choo-choo train just broke

My name is erich, this choo-choo train is so fun :-)
My name is katie, this choo-choo train is so fun :-)

```

If we want the instance variable `myToy` to reference memory (a toy) allocated elsewhere, such as in `main`, we have two choices: use a reference variable or use a pointer.

### 12.1.4 Reference Variables

We can achieve the desired behavior by adding one character to the code in Program 12.2. If we change the declaration of `Toy myToy` to `Toy& myToy` the output changes as follows.

## O U T P U T

```

prompt> sharetoy
My name is erich, this choo-choo train is so fun :-)
My name is katie, this choo-choo train is so fun :-)

oops, this choo-choo train just broke

My name is erich, this choo-choo train is broken :-)
My name is katie, this choo-choo train is broken :-)

```

Just as a reference parameter is an alias for memory allocated elsewhere, a reference variable refers to memory allocated elsewhere. In Program 12.2, making `myToy` a reference variable avoids creating a copy when `myToy` is initialized in the `Kid` initializer list. Instance variables that are references *must* be constructed and initialized using an initializer list, not in the body of the class constructor. Once a reference variable is constructed, it cannot be re-assigned to. In Program 12.2, *sharetoy.cpp*, this means that a `Kid` object cannot change toys; it's impossible to add new member functions that change the toy a kid uses for play.

### 12.1.5 Pointers for Sharing

In some situations we'd like to change the object being shared, that is, change the toy shared in Program 12.2. Furthermore, it's not possible to have vectors of references in C++ so we must turn to pointers. Using pointers for sharing allows a shared object to be changed and makes it possible to use a vector to share many objects.

We'll develop a program for several walkers to record their movements with a `WalkRecorder` object shared among the walkers. We'll keep the example simple to illustrate the concept and to highlight a problem that arises frequently when several interdependent classes are used in the same program. The problem arises when class A uses class B and *vice versa*. This interdependency can create compilation problems if you don't design the class interfaces properly and write the header files with care. We'll discuss the design of the walker program, then the problems with interdependencies, then we'll show the program implementation that addresses the interdependency problems.

We'll use two classes in the program:

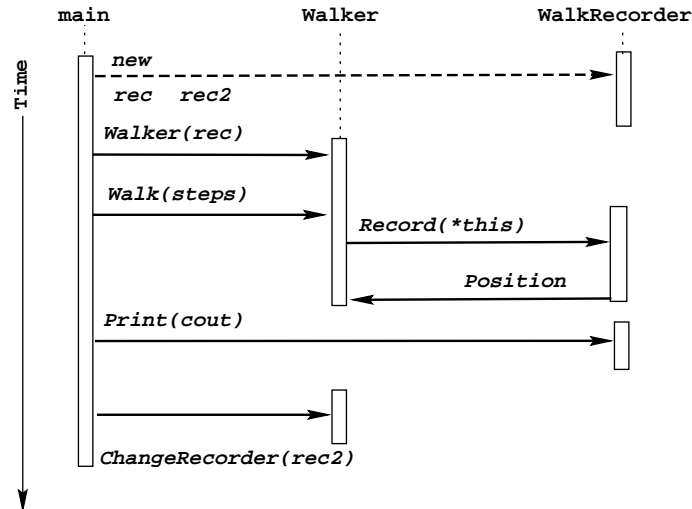
- The class `Walker` simulates a one-dimensional walker recording the walk with a `WalkRecorder` object.
- The class `WalkRecorder` records a walker's position. A walker is passed to the recorder and the recorder then queries the walker to get its position to record it.

To record a walk, each walker must know about a `WalkRecorder` object. We'll design the `Walker` class so that each walker object maintains a pointer to the recorder that's recording the walker's movements. It will be possible to share a recorder among several walkers or to give each walker a separate recorder object. In designing the classes and programs we must consider at least three questions.

- A `WalkRecorder` records a walker. Who is responsible for passing the walker? How is the walker passed to the `WalkRecorder`?
- Where are `Walker` and `WalkRecorder` objects created? In `main`? Does a walker create its own recorder?
- How is the data recorded by a `WalkRecorder` displayed?

To keep the program simple we'll create all the objects in `main`. In a more complex program you might create a class in charge of object creation. We'll create a recorder, then pass the recorder to each `Walker` object when the `Walker` is created, but we'll also design a method for changing a walker's recorder.

Since a walker knows its recorder, we'd like the walker to ask the recorder to make a record of the walker itself. Each walker can pass itself to its recorder using the reserved word **this** which every object has as a pointer to itself. A variable named `foo` in `main` might be known as the parameter `firstFoo` in a function to which it's passed as an argument. In general, objects have different names in different places in a program. However, in C++ every object uses the identifier `this` as its own name. Because `this` is a pointer, `*this` is the way an object identifies itself since "star this" is also "the object pointed to by this" which is itself!



**Figure 12.3** Interaction diagram for the classes `Walker` and `WalkRecorder` showing a recorder being shared and changed.

Finally, the code in `main` will ask a recorder to print the data the recorder has kept track of. Again, in a more complex program we might provide member functions for retrieving the data, but for now we'll be content with printing the recorded data.

A first draft of the two classes is shown in Program 12.3. The interactions between these classes and the `main` of Program 12.4, `frogwalk3.cpp` are shown in the **interaction diagram** in Figure 12.3. As a program executes, time increases from the top of the diagram to the bottom. Arrows indicate when one class (or program segment) calls another, and the method used to make the call. The dashed line at the top of the diagram indicates an indirect call of a constructor via `new`.

### 12.1.6 Interdependencies, Class Declarations, and Header Files

We're ready to start implementing the classes. Although the final program `frogwalk3.cpp` shows everything in one file, we'll discuss the class declarations and definitions assuming that separate `.h` and `.cpp` files are used. A high-level first pass yields `Walker` on the right and `WalkRecorder` on the left.

#### Program 12.3 `walkdesign.cpp`

```

#ifndef _WALKRECORDER_H
#define _WALKRECORDER_H

#include "walker.h"

#endif

```

```

#ifndef _WALKER_H
#define _WALKER_H

#include "walkrecorder.h"

#endif

```

```

class WalkRecorder
{
public:
    WalkRecorder();
    void Record(const Walker& walker);
    void Print(ostream& out) const;
private:
    tmatrix<int> myRecord
};

class Walker
{
public:
    Walker(WalkRecorder* wrec);
    void Walk(int steps);
    int Position() const;
    void ChangeRecorder(WalkRecorder* wrec);
private:
    int myPosition;
    WalkRecorder * myRecorder;
};

```

walkdesign.cpp

This design won't compile when we create the main program that includes both header files. Remember that the preprocessor (see Section 7.2.3) literally cuts and pastes a .h file when a `#include` is processed, and that include files that are included by an include file are also cut-and-pasted (and include files that they include and so on.) This is why the `#ifndef _CLASSNAME_H` appears at the top of each include file. Without this, consider the following main program that includes both classes.

```

#include "walker.h"
#include "walkrecorder.h"
int main()
{
    // ...
    return 0;
}

```

Since `walker` includes `walkrecorder` which includes `walker` which includes ... there would be an infinite chain of cut-and-paste includes without the protecting `#ifndef` statements. These protecting statements do stop an infinite chain of includes, but there's a different problem.

The line `#include "walker.h"` appears in `walkrecorder.h` (as simulated and shown in `walkdesign.cpp`, Program 12.3) because the class `Walker` is used as a parameter in `WalkRecorder::Record`. Similarly the class `WalkRecorder` is a parameter in two `Walker` methods. However, if the main program above is used, where the first include is `#include "walker.h"`, then the preprocessor creates the following compilation unit.

```

class WalkRecorder
{
    ...
};
class Walker
{
    ...
};
// more code here

```

The classes appear in the order shown, with `WalkRecorder` first, because the preprocessor first processes `walker.h`. The first line in this header file is another `#include` so

this include for *walkrecorder.h* is processed before the declaration of the class `Walker` is read by the preprocessor. The include file in *walkrecorder.h* isn't a problem, it's not preprocessed because of the `#ifndef` protection, but the class `WalkRecorder` does appear first when the compiler is called after the preprocessor finishes.

The compiler stops at the declaration of the method `WalkRecorder::Record` because the compiler hasn't yet seen the class `Walker`, so it doesn't know anything about the parameter! How can this problem be fixed? This problem is fixable, but only because the classes make reference to each other in the class declarations using only references or pointers to the other class. The compiler doesn't need to know the names of `Walker` member functions or how big a `Walker` object is to compile the header file for `WalkRecorder`. Similarly, the header file for `Walker` can be compiled without knowing the details of `WalkRecorder`. Suppose, however, that the instance variable `myRecorder` isn't a pointer, but is declared as follows.

```
WalkRecorder myRecorder;
```

With this declaration we won't be able to share recorders since a `Walker` object's recorder will be a copy (see Program 12.2). In addition, the compiler must know how much memory a `WalkRecorder` requires (the sum of the sizes of its instance variables) to compile this declaration. Because all pointers and references are basically aliases or indirect references to memory allocated elsewhere, all pointers and references use the same amount of memory, regardless of the type of object being pointed to or referenced. If the class declaration in a header file uses another class with only pointers or references, it's possible to create a **forward reference** to the class being used. For *walker.h* the forward reference of `WalkRecorder` looks like this.

```
class WalkRecorder;
class Walker
{
public:
    Walker(WalkRecorder* wrec);

    void Walk(int steps);
    int Position() const;
    void ChangeRecorder(WalkRecorder* wrec);

private:
    int myPosition;
    WalkRecorder * myRecorder;
};
```

Now the preprocessor won't have any problems. The compiler parses the forward reference of `WalkRecorder` as a class whose declaration will be supplied later. Later is good enough since the compiler doesn't need to know the names of `WalkRecorder` methods nor how big a `WalkRecorder` object is to compile the `Walker` declaration.

586

## Chapter 12 Dynamic Data, Lists, and Class Templates

In the implementation file, *walker.cpp*, you'll need to write

```
#include "walkrecorder.h"
```

since the implementation of `Walker::Walk` calls `myRecorder->Record` as shown in the interaction diagram Figure 12.3. This isn't a problem though because the header files don't include each other, they're simply included in a `.cpp` file as needed. The complete Program 12.4 shows forward declarations, class declarations and implementations. A run follows the program listing.

**ProgramTip 12.2: Use forward references rather than `#includes` whenever possible in a header file.** If a class `Foo` uses a class `Thing` and `Thing` objects appear only as pointers or references in parameters and instance variables, then the header file *foo.h* should use `class Thing;` as a forward reference rather than `#include "thing.h"`.

---

**Program 12.4** *frogwalk3.cpp*

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

#include "prompt.h"
#include "tvector.h"
#include "randgen.h"
#include "dice.h"

class Walker;
class WalkRecorder
{
public:
    WalkRecorder();
    void Record(const Walker& walker);
    void Print(ostream& out) const;

private:
    static int MAX;
    tvector<int> myRecord;
    int myBeyondCount;
};

class Walker
{
public:
    Walker(WalkRecorder* wrec);

    void Walk(int steps);
```

## 12.1 Pointers as Indirect References

587

```
    int Position() const;
    void ChangeRecorder(WalkRecorder* wrec);

private:
    int myPosition;
    WalkRecorder * myRecorder;
};

int WalkRecorder::MAX = 100;

WalkRecorder::WalkRecorder()
    : myRecord(2*MAX+1,0), myBeyondCount(0)
{
    // record -MAX..MAX, all zero
}

void WalkRecorder::Record(const Walker& walker)
{
    int pos = walker.Position();
    if (fabs(pos) > MAX)
    {   myBeyondCount++;
    }
    else
    {   myRecord[pos+MAX]++;
    }
}

void WalkRecorder::Print(ostream& out) const
{
    int lowIndex=-1,highIndex=0,k;
    for(k=0; k < 2*MAX+1; k++)
    {   if (myRecord[k] != 0)
        {   if (lowIndex == -1) lowIndex = k;
            highIndex = k;
        }
    }
    if (lowIndex == -1)
    {   out << " no steps taken" << endl;
        return;
    }
    for(k=lowIndex; k <= highIndex; k++)
    {   cout << k-MAX << "\t" << myRecord[k] << endl;
    }
    cout << endl << "beyond boundaries = " << myBeyondCount << endl;
}

Walker::Walker(WalkRecorder * wrec)
    : myPosition(0), myRecorder(wrec)
{
}

void Walker::Walk(int steps)
{
    Dice d(2);
```

588

## Chapter 12 Dynamic Data, Lists, and Class Templates

```
int k;
for(k=0; k < steps; k++)
{
    if (d.Roll() == 1)
    {
        myPosition++;
    }
    else
    {
        myPosition--;
    }
    myRecorder->Record(*this);
}
}

int Walker::Position() const
{
    return myPosition;
}

void Walker::ChangeRecorder(WalkRecorder* wrec)
{
    myRecorder = wrec;
}

int main()
{
    WalkRecorder * rec = new WalkRecorder();
    WalkRecorder * rec2 = new WalkRecorder();
    Walker w1(rec);
    Walker w2(rec);
    int steps = PromptRange("how many steps ",1,10000);
    w1.Walk(steps);
    w2.Walk(steps);
    rec->Print(cout);

    cout << endl << "another walk" << endl << endl;

    w1.ChangeRecorder(rec2);
    w2.ChangeRecorder(rec2);
    w1.Walk(steps);
    w2.Walk(steps);
    rec2->Print(cout);
    return 0;
}
```

frogwalk3.cpp

```
OUTPUT

prompt> frogwalk3

how many steps between 1 and 10000: 20
-6      1
-5      2
-4      3
-3      5
-2      7
-1      7
0       5
1       6
2       4

beyond boundaries = 0

another walk

-1      1
0       2
1       5
2       7
3       6
4       6
5       5
6       4
7       3
8       1

beyond boundaries = 0
```

### 12.1.7 Delete and Destructors

Whenever a program allocates memory from the freestore (heap) using `new`, the memory should eventually be returned to the freestore using `delete` when the memory is no longer needed. For example, at the end of `main` in Program 12.4 we could add the two lines shown here.

```
int main()
{
    WalkRecorder * rec = new WalkRecorder();
    WalkRecorder * rec2 = new WalkRecorder();
```

```

...
delete rec;
delete rec2;
return 0;
}

```

Returning the `WalkRecorder` objects referenced by pointers `rec` and `rec2` to the heap isn't really necessary here since all memory used by a program is reclaimed by the system when the program terminates. The `delete` operator returns memory allocated by `new`; it takes a pointer as an argument. Although the argument to `delete` is a pointer,

#### Syntax: The `delete` operator

```
delete ptr;
```

an object is returned to the heap, not the pointer used in the statement when `delete` is called. The pointer must point to an object allocated by `new` or an error will occur. If

you delete a stack object, for example, the system may think the object came from the heap and will be reused in a subsequent call of `new`.

This almost always causes trouble in a program. Similarly, you should not delete an object twice since the system's bookkeeping may think the object is free twice, but there is only one object, not two.

```

Date * dptr = new Date(); // today
delete dptr;             // ok, reclaim memory
delete dptr;             // trouble, reclaimed twice

```

Deleting an object *does not* change the value of the pointer to the object, but the pointer is now referencing memory that is no longer valid having been returned to the freestore. It is also an error to dereference a pointer immediately after the object it points to has been deleted:

```

Date * dptr = new Date(); // today
delete dptr;             // ok, reclaim memory
Date tomorrow(*dptr + 1); // trouble, *dptr doesn't exist

```

The code above may seem to work when you run it, but this style of programming will eventually lead to an error that's very difficult to track down. Some programmers assign the special pointer value zero to a pointer after deleting the object it points to.

```

Date * dptr = new Date(); // today
delete dptr;             // ok, reclaim memory
dptr = 0;                // errors easier to find
delete dptr;             // ok, no memory
Date tomorrow(*dptr + 1); // immediate error caused

```

A pointer with the value zero is called a **null pointer**. In C++ you can write `p = NULL` where `p` is a pointer, but the identifier `NULL` is not a reserved word, it's a preprocessor macro defined in the standard header file `<cstdlib>` which is almost always included by some other standard header file. It's better to use zero since no header files are needed. Dereferencing a null pointer causes an immediate error, a segmentation fault on

Unix/Linux systems, a general protection fault or unhandled exception on other systems. Immediate errors are good because you can almost always find the cause of the error using a symbolic debugger. It is not an error to delete a null pointer, so assigning zero to a pointer after deleting the object it points to is a reasonable defensive programming strategy.

In general you should try to return memory no longer needed to the heap or it's possible your program will eventually use up all the memory available. Consider the following code.

```
Date today;
Date * dptr;
while (true)
{
    int month = PromptString("enter month (0 to exit)", 0,12);
    if (month == 0) break;
    dptr = new Date(1,month,today.Year());
    MakeCalendar(*dptr);
    // should call delete dptr here, but forgot
}
```

Each time the loop iterates a `Date` object is allocated from the heap. The next time the loop iterates, the previous `Date` object is lost; there is no pointer referencing it so it has become inaccessible. If the loop executes 10,000 times then 10,000 `Date` objects will have been allocated and remain unusable. In some languages, like Java, memory that is no longer accessible is automatically reclaimed using a technique called **garbage collection**. In standard C++ environments there is no automatic garbage collection, programmers are responsible for it.

**Program Tip 12.3: Deleting objects is a good idea, but deleting improperly will cause problems in your program.** You can't, for example, delete an object twice without eventually causing problems. Nor can you delete an object that wasn't allocated using `new` without causing problems. When you're developing a program, add `delete` code only when you know your program is working correctly so that any error due to improper deletes can be found without looking at other code.

**The Destructor Member Function.** Just as a constructor is called automatically when an object is defined; a special member function called the **destructor** is called automatically when an object goes out of scope.<sup>3</sup> When an class object allocates memory, the object should be responsible for deleting the memory. If the memory is referenced by an instance variable it cannot be deleted until the object is no longer needed. The destructor will be called either when the program deletes the object using `delete` or when the object goes out of scope and isn't accessible. For any class named `Thing`, a member

<sup>3</sup>You can think of *going out of scope* as becoming undefined to contrast with definition and the constructor.

function named `~Thing` is the class **destructor**. We'll discuss destructors in more detail in Section 12.10

## Pause to Reflect



**12.10** Assume that a reference instance variable `Toy& myToy` is used in the class `Kid` as described in Section 12.1.4. The function `MakeKid` returns a pointer to a `Kid` object as follows.

```
Kid * MakeKid()
{
    Toy block("wooden block");
    Kid * kptr = new Kid("alex",block);
    return kptr;
}
```

Explain why the object pointed to and returned by `MakeKid` will cause problems.

**12.11** If the instance variable `myToy` is changed to a pointer, how do the member functions of the class `Kid` change?

```
class Kid
{
    ...
private:
    Toy * myToy;
};
```

**12.12** Write declarations and implementations of all methods of a modified `Kid` class. Each `Kid` creates his/her own toy allocated from the heap and stores a pointer to the toy. Three methods are added: `GetToy`, `ShareFrom`, and `Unshare`. The functions are used as follows.

```
Kid robert("robert"); // creates his own toy
Kid laura("laura"); // creates her own toy
laura.Play(); // play with own toy
robert.Play();

robert.ShareFrom(laura); // robert shares laura's toy
robert.Play(); // with laura's toy
robert.Unshare();
robert.Play(); // with robert's original toy
```

The function `GetToy` is called in the implementation of `ShareFrom`. Can `GetToy` be private?

- 12.13** Using forward references (see Program Tip 12.2) rather than `#include` statements can save on preprocessor time and make it less necessary to recompile a client program when classes the client uses are changed. Consider the program fragment in the previous exercise that shows two `Kid` objects playing. Explain why it is necessary to have `#include "kid.h"` in the program above, but it is *not* necessary to have `#include "toy.h"`. Explain why `class Toy` can be a forward reference in `kid.h` but why `#include "toy.h"` is needed in `kid.cpp`. Finally, explain why the client code above does *not* need to be compiled if the implementation of `Toy` changes, but why `kid.cpp` will need to be recompiled, and why the program must be relinked.
- 12.14** Create an interaction diagram for the code fragment above in which two `Kid` objects play and share a toy. Show `main`, `Kid`, and `Toy`. Include details about when/where objects are created and when/where all member functions are called.
- 12.15** Design a class `ToyChest` that holds several pointers to toy objects. `Kids` should be able to get toys from the chest and put toys back in the chest. Consider at least two ways to have toys added to the chest: when constructed the chest creates its own toys; and a `Kid` can add a toy to a chest that originated in a different toy chest. You'll need to think carefully about the design so that a toy can be shared among kids playing with it, but reside in only one toy chest.
- 12.16** If a `Kid` allocates his/her own toy from the heap, who is responsible for deleting the toy?
- 12.17** In `frogwalk3.cpp`, Program 12.4, a new recorder is attached to the `Walker` objects in `main`. Write a new member function `WalkRecorder::Clear()` that clears a recorder's memory. Show how to use this new function to achieve the same effect of `frogwalk3.cpp`, but using only one recorder that's cleared rather than using two recorders.
- 12.18** When should the `WalkRecorder` objects in `frogwalk3.cpp` be deleted?
- 12.19** How can the class `WalkRecorder` be changed to track every position, not just those between `-MAX` and `MAX`?
- 12.20** What is the purpose of the loop in `WalkRecorder::Print`? Why is the value of `lowIndex` compared to `-1`?
- 12.21** Write code to create a vector of 100 pointers to `Dice` objects, making `a[k]` point to a  $(2k + 1)$ -sided `Dice`. Roll each `Dice` 1000 times, then delete all the objects.

### Alan Perlis (1922–1990)

In 1966 Alan Perlis became the first recipient of the Turing award. The award was given for his work in programming language design. In 1965 he estab-



lished the first graduate program in computer science at what was then the Carnegie Institute of Technology and is now Carnegie-Mellon University.

In [AS96], Perlis is quoted with some important advice to novices and experts in computer science: “*I think that it’s extraordinarily important*

*that we in computer science keep fun in computing. ... I hope the field of computer science never loses its sense of fun. ... What’s in your hands, I think and hope, is intelligence: the ability to see the machine as more than when you were first led up to it, that you can make it more.*”

In his Turing award address, Perlis looked ahead to parallel and distributed computation, a field that has been growing steadily and receiving increased attention in recent years. He also talked of the intellectual foundation of programming, from Turing’s work to the languages LISP and ALGOL, which have had a profound impact on programming language design.

In [AS96] he writes about programming:

*To appreciate programming as an intellectual activity in its own right you must turn to computer programming; you must read and write computer programs—many of them. It doesn’t matter much what the programs are about or what applications they serve. What does matter is how well they perform and how smoothly they fit with other programs in the creation of still greater programs.*

A list of Perlis epigrams has been gathered; these include:

- Most people find the concept of programming obvious, but the doing impossible.
- Once you understand how to write a program, get someone else to write it.
- The best book on programming for the layman is *Alice in Wonderland*; but that’s because it’s the best book on anything for the layman.

For more information see [Per87]

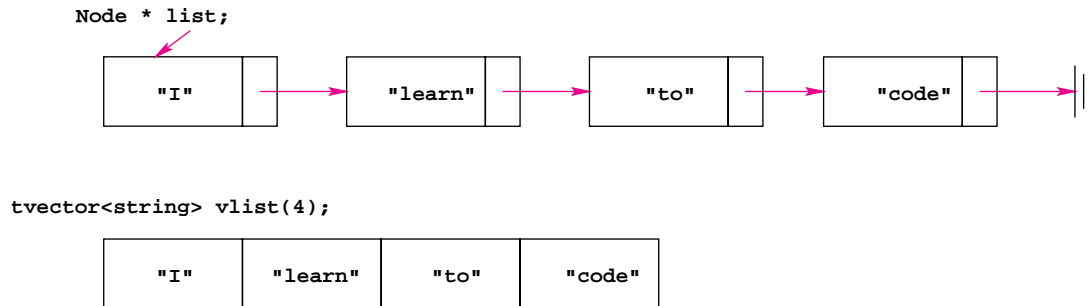


Figure 12.4 Comparing vectors and linked lists.

## 12.2 Linked Lists

A **linked list** stores a sequence of items as does a vector. However, vectors support random access to any element: the time needed to access `a[5]` is the same as the time needed to access `a[100]` when `a` is a `tvector`. In contrast, items that are stored near the front of a linked list are accessed more quickly than items near the end of a linked list. This is analogous to how the songs, or tracks, on a cassette tape are arranged. Accessing the fifth song requires skipping over the first four, and songs near the end of the tape take longer to access than songs near the front. Arrays are more like compact discs; it's as easy to play the last track of a CD as it is the first because CD players provide random access to the tracks.

Like any recording tape, linked lists permit new items to be “spliced” into the middle of a list. In the same way that tapes can become longer by splicing in new segments of tape and can be made shorter by cutting out segments of tape, linked lists can have items added and deleted from any location in the list without shifting other items in the list. In some sense, pointers link together the different items of a list in the same way that glue or tape is used to splice segments of magnetic tape.

If a vector contains 100 items, the items must be allocated contiguously in memory. When linked lists are used, the different items (these items are usually called **nodes** in contrast with the cells of a vector) do *not* need to be allocated contiguously. Each node of a linked list has a pointer to the node that follows it; these pointers are the “tape” used to splice nodes together. Figure 12.4 shows a linked list and an array that store the same values. The last node of a list usually points to 0 (NULL) so that a program can determine when the last node has been reached. This is diagrammed in Figure 12.4 with the symbol for an electrical ground, three vertical bars.

Abstractly, a linked list is very similar to a `CList` object, and in fact linked lists are used to implement the `CList` class. A linked list has a first node, like the `Head` of a `CList`. The first node points to all the other nodes in the linked list, specifically to the first of these other nodes. The other nodes are like the `Tail` of a `CList` object. More concretely, a node in a linked list contains the information stored in the node, and

a pointer to the next node in the list. In C++ a node storing a string is declared like this:

```
struct Node
{
    string info;
    Node * next;
};
```

The `info` field of the `struct` stores information, in this case a string. The `next` field stores a pointer to the next node in the list. This declaration is self-referential: the declaration for `Node` includes a pointer to a node. This is fine because a pointer can be declared without knowing completely how much memory the thing it points to uses as we saw in Section 12.1.6. It would be illegal, for example, to declare `Node` as

```
struct Node
{
    string info;
    Node next;
};
```

Here the `next` field isn't a pointer, but a `Node`. This declaration is circular and will be rejected by the compiler. The `g++` compiler generates an error message (in a program named `foo.cpp`):

```
foo.cpp:4: field 'next' has incomplete type
```

When the compiler parses the declaration for `next`, the declaration for the `struct Node` is not yet complete. The declaration can be incomplete for a pointer to a `Node` to be used, but not for a `Node`.

Program 12.5, `strlink.cpp`, shows how a `tvector` and a linked list are initialized to contain the four strings "I," "learn," "to," "code." A `tvector` of strings is stored in the variable `vec` and a linked list based on the `struct Node` is pointed to by a pointer variable `first`. When you write code that uses linked lists you'll need to maintain a pointer to the first node in the list. Often, you'll need to maintain a pointer to the last node to make it easier to add a node at the end of the list. You could write code to find the last node by starting at the beginning and traversing the list until the last node is found (the `next` field of the last node points to 0). It's much faster, however, to maintain pointers to both the first and last nodes. Only one pointer, to the first node of a list, is maintained in `strlink.cpp`.

---

#### Program 12.5 `strlink.cpp`

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

// compare linked list construction to vector construction
```

```
struct Node
{
    string info;
    Node * next;
    Node(const string& s, Node * link)
        : info(s),
          next(link)
    { }
};

void Print(Node * list);
void Print(const tvector<string> & list);

int main()
{
    Node * first=0;    // initially no nodes in list
    Node * temp=0;    // initialize to 0 for defensive programming
    int k;
    tvector<string> vec;
    string storage[] = {"I", "learn", "to", "code"};

    for(k=0; k < 4; k++)
    {
        vec.push_back(storage[k]);
        temp = new Node(storage[k],first); // new node before first
        first = temp;                    // make first point at new node
    }
    cout << "vector:\t\t";
    Print(vec);
    cout << "linked list:\t";
    Print(first);
    return 0;
}

void Print(Node * list)
// pre: list is 0-terminated (last node's next field is 0)
// post: all info fields of list printed on one line
{
    Node * temp;
    for(temp = list; temp != 0; temp = temp->next)
    {
        cout << temp->info << " ";
    }
    cout << endl;
}

void Print(const tvector<string> & list)
// pre: list contains list.size() entries
// post: all elements printed on one line
{
    int k;
    for(k=0; k < list.size(); k++)
    {
        cout << list[k] << " ";
    }
    cout << endl;
}
```

## O U T P U T

```
prompt> strlink
vector:      I learn to code
linked list: code to learn I
```

### 12.2.1 Creating Nodes with Linked Lists

The values stored in the built-in array `storage` are used to initialize the vector `vec` and the linked list pointed to by `first`. Since new values are added to the front of the list, the values are in reverse order from those stored in the vector as the output shows. To add new nodes at the end of the list requires maintaining a pointer to the last node. This is straightforward except for a fencepost problem of creating the first node. As we'll see, it's easy to add or splice-in a new node after another node. Initially there are no nodes, so there's no last node to add a new node after. We'll need special case code to deal with this fencepost problem or we'll need to use a header node as discussed in Section 12.2.6.

**Program Tip 12.4:** Creating a linked list often requires special-case code to manage the creation of the first node since this node is the only node that doesn't follow another node. Sometimes creating a dummy first node (called a header node) avoids lots of special-case code.

Since there's a constructor for the struct `Node`, it's simple to create a new node and initialize its fields. If there were no constructor we'd need three statements to allocate and initialize a node.

```
temp = new Node();           // create new node
temp->info = storage[k];     // store info
temp->next = first;         // point at first node of list
```

The two statements creating the node and ensuring that `first` points at the new node can be combined into a single statement.

```
first = new Node(storage[k],first); // new first node
```

The "old" value of `first` is used to construct the node, so the new node points at the old first node. The pointer to the newly created node is returned by `new`, and the pointer value is assigned to `first` creating a new first node. This statement mirrors exactly the use of `cons` with a `CList` object for adding a new node to the front of a list.

```
CList<string> list;
list = cons(string("apple"), list);
```

### 12.2.2 Iterating over a Linked List

The `for` loop that prints all the nodes of the linked list in the function `Print` of Program 12.5, *strlink.cpp*, is the standard method for looping over all nodes of a linked list. If `list` points at the first node, we write:

```
Node * temp;
for(temp = list; temp != 0; temp = temp->next)
{
    // process *temp
}
```

The statement `temp = temp->next` advances the pointer `temp` so that it points at the next node, (e.g., at the second node if it used to point to the first node). When the loop finishes, `temp` is zero, or `NULL`. Because `list` is passed by value to `Print`, changes to `list` don't affect the argument passed. Since the parameter is a copy, we don't need the temporary pointer and could write the following loop instead.

```
for( ; list != 0; list=list->next)
{
    cout << list->info << " ";
}
```

There's no initialization in the `for` loop because `list` points at the first node.

Many programmers prefer to use a `while` loop for iterating over a list.

```
while (list != 0)
{
    // process *list
    list = list->next;
};
```

There's nothing inherently wrong with using the temporary pointer, and we'll see that a temporary pointer is often required in a class-based use of linked lists.

Of course `Print` can be written recursively too.

```
void Print(Node * list)
{
    if (list != 0)
    {
        cout << list->info << " ";
        Print(list->next);
    }
}
```

The recursive function doesn't insert an `endl` onto the stream. This would be done in the client code that calls the recursive `Print`.

**Program Tip 12.5:** When a node-pointer is passed by value, changes to the pointer cannot affect the pointer passed as an argument, but changes can be made to the object pointed to; these changes have an effect on the node. The distinction here is between the pointer, which is passed by value, and the node, which isn't really passed, but the pointer to the node can be used to change the node. In other words, `list = NULL` doesn't affect a pointer parameter `list` passed by value, but `*list = ...` does affect the object pointed to.

### 12.2.3 Adding a Last Node to a Linked List

We'll discuss modifications to *strlink.cpp* that add new nodes to the end of the list instead of the front. We'll use a pointer named `last` that always points to the last node of the list being constructed, so the statement "last points to last node in linked list" is a loop invariant. Since the invariant must be true the first time the `for` loop test is evaluated, we must create a last node before the loop. Initializing `last = 0` does *not* create a node, so we must create a node before the loop. We have two choices in creating an initial node.

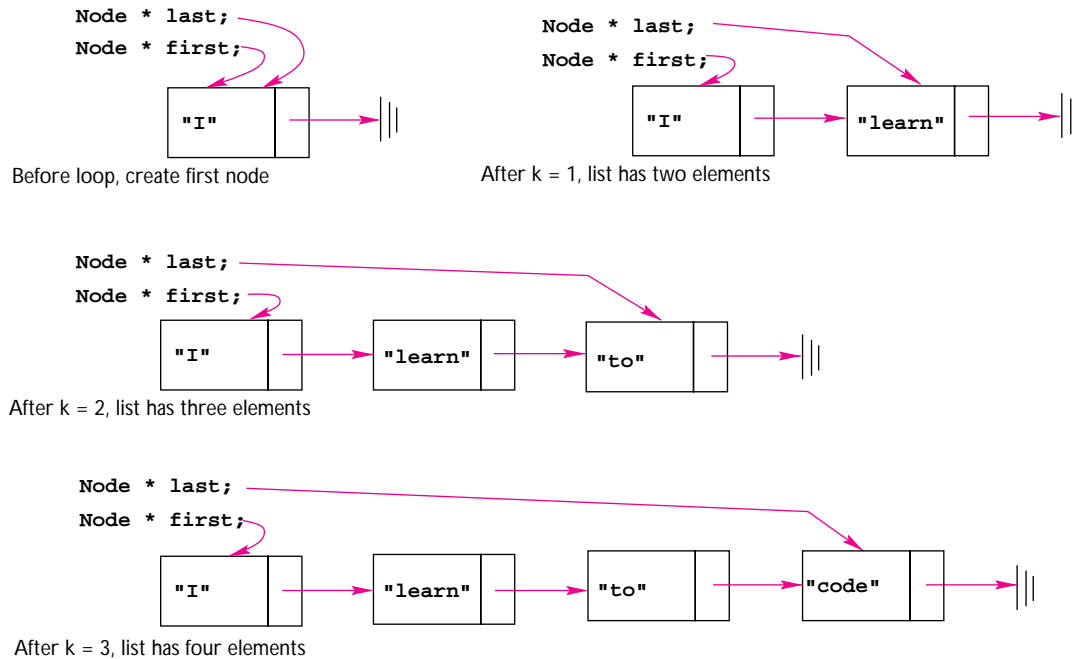
- Store data in the node so that the initial last node is also the initial first node storing "I" in Program 12.5. This is the approach used in the program fragment below. Since we use the same loop for creating nodes and vector elements, we would need to add a value to the vector before the loop too.
- Create a dummy, also called a header, node that does not store data, but is used so that even the first node in a list has a node before it. With a header node, every node in the list has a predecessor node (the header node isn't considered part of the list.)

The formation of the linked list at each iteration of the loop is diagrammed in Figure 12.5. Note that after each loop iteration the variable `last` points to the last node of the linked list. The variable `first`, initialized before the loop because of the fencepost problem, never moves and always points to the first node of the linked list.

```
Node * first = 0;
Node * last = 0;
last = first = new Node(storage[0],0); // last is first
for(k=1; k < 4; k++)
{   last->next = new Node(storage[k],0); // new last node
    last = last->next;                  // update last
}
```

### 12.2.4 Deleting Nodes in a Linked List

The nodes allocated in Program 12.5, *strlink.cpp*, are not deleted. We'll write a function `DeleteNodes` to delete all the nodes in a linked list whose first node is passed to the

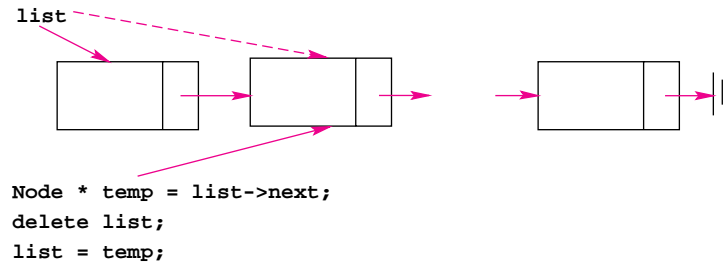


**Figure 12.5** Building a linked list by adding a last node.

function. Deleting nodes in a linked list requires careful coding. We'll use recursion in `DeleteNodes` because it's much easier than writing a loop. As with all recursive functions, some base case must be identified. When linked lists are used, the base case is usually the empty list (typically a `NULL`/zero pointer) although sometimes a one-node list can be used as a base case.

```
void DeleteNodes(Node * list)
// post: all nodes in list are deleted
{
    if (list != 0)
    {
        DeleteNodes(list->next); // delete after me
        delete list;           // delete first node
    }
}
```

If you believe that the recursion handles all nodes after the first node, then the function works as intended since after deleting all the other nodes, the first-node is returned to the freestore. Writing an iterative version of this function requires a temporary pointer as illustrated in Figure 12.6.



**Figure 12.6** Deleting the first node of a linked list.

Since the first node will be deleted, we must initialize a temporary pointer `temp` to point to the second node. After deleting the first node, the pointer `list` can be reassigned to point to the second node whose value was saved in `temp`.

```

void DeleteNodes(Node * list)
// post: all nodes in list are deleted
{
    Node * temp;
    while (list != 0)
    {
        temp = list->next; // remember next node in list
        delete list;     // first node gone
        list = temp;     // new first node
    }
}

```

At first, you might think that a temporary pointer isn't necessary and that the following code can be used to delete the first node pointed to by `list`:

```

delete list;
list = list->next;

```

There is a problem with this code: you can't be sure what happens to the node pointed to by `list` after the deletion. Once deleted, the node is garbage and may be reclaimed by some other program or some other part of the system. Some programming environments may explicitly fill all deleted storage with garbage. In these cases, dereferencing `list` using `list->next` can result in a bad dereference, causing the program to abort. Although your code has not done anything with the storage that `list` used to point to, which was just deleted, you cannot be sure that the node still exists or that the `next` field has the same value. You must use a temporary variable.

### 12.2.5 Splicing Nodes into a Linked List

One of the primary advantages of using a linked list instead of a vector is the ability to add new nodes to the middle of a list without shifting the other nodes. To add a new

value to a sorted vector we must shift the vector elements to make room for the new element. No shifting is required to add a new node to a sorted linked list so that the list remains sorted. Program 12.6 shows a function `AddInOrder` that inserts a new string into a sorted linked list of strings keeping the list sorted. We'll discuss several different implementations of `AddInOrder`.

---

Program 12.6 `orderedlist.cpp`

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

// read words in a file, store in order in a linked list

struct Node
{
    string info;
    Node * next;
    Node(const string& s, Node * link)
        : info(s), next(link)
    { }
};

Node* AddInOrder(Node* list, const string& s)
// pre: list is sorted
// post: add s to list, keep list sorted, return new list with s in it
{
    Node * first = list; // hang onto first node

    // if new node is first, handle this case and return
    if (first == 0 || s < first->info)
    {
        return new Node(s,first);
    }

    // assert: s >= list->info
    while (list->next != 0 && list->next->info < s)
    {
        list = list->next;
    }
    // assert: s >= list->info and s < list->next->info (conceptually)

    list->next = new Node(s,list->next);
    return first;
}

void Print(Node * list)
{
    for(; list != 0; list=list->next)
    {
        cout << list->info << endl;
    }
}
```

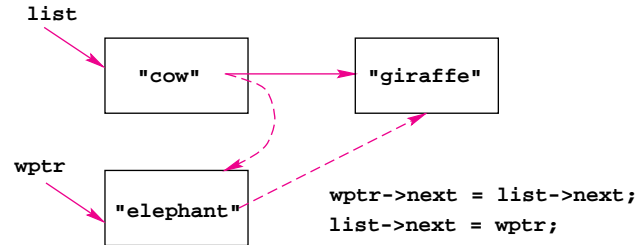


Figure 12.7 Adding a new node to a sorted linked list.

```

int main()
{
    Node * list = 0; // empty
    string word, filename = PromptString("filename: ");
    ifstream input(filename.c_str());
    while (input >> word)
    { list = AddInOrder(list,word);
    }
    Print(list);
    return 0;
}

```

orderedlist.cpp

## OUTPUT

```

prompt> poe.txt
!ugh!-ugh!
A
2321 words not shown
your

```

To add a new node in order using iteration we must maintain a pointer to the node before where the new node goes. In Figure 12.7 a new node containing "elephant" is added to a sorted linked list. In the diagram, the new node is pointed to by `wptr` and the node is added after the "cow" node pointed to by `list`. To search for the location to add the new node we must look one node ahead. For example, we don't know that "elephant" goes after "cow" until we know "giraffe" follows cow. If "dog" follows "cow", we need to keep searching.

A recursive version of the function `AddInOrder` from Program 12.6 is simpler than the iterative version. Note that the base case is also handled in the original version of `AddInOrder`.

```

Node* AddInOrder(Node* list, const string& s)
// pre: list is sorted
// post: add s to list, keep list sorted, return new list
{
    if (list == 0 || s < list->info) // new node goes first
    {
        return new Node(s,list);
    }
    list->next = AddInOrder(list->next,s);
    return list;
}

```

The base case handles an empty list or a list in which all strings are greater than the string being added. The order of the boolean tests is important. If the test for `s < list->info` is made first, the test will cause an error when `list == 0` since a NULL/zero pointer will be dereferenced.

**Program Tip 12.6: Every pointer dereference should be guarded either explicitly by a check that the pointer is not NULL/zero or implicitly by documenting and reasoning that the pointer cannot be zero.** Each time you write code of the form `list->data_member` you should either check `list != 0` before the dereference or be able to show with formal reasoning that `list` cannot be NULL/zero.

We can change the function `AddInOrder` so that `list` is passed by reference. It may be harder to see that this function is correct.

```

void AddInOrder(Node* & list, const string& s)
// pre: list is sorted
// post: add s to list, keep list sorted
{
    if (list == 0 || s < list->info) // new node goes first
    {
        list = new Node(s,list);
    }
    else
    {
        AddInOrder(list->next,s);
    }
}

```

Two observations may help you see that this version of `AddInOrder` works correctly.

1. The base case correctly changes `list` when the list is empty or when the new node belongs before all other nodes. The base case creates a node that points to the old first node and makes `list` point to the new node. Since `list` is passed by reference, the change is propagated back to the calling statement.
2. In each recursive call, the argument `list->next` is passed by reference. This

means that each clone called recursively uses the name `list` as an alias for some `next` field of the linked list being processed.

You may need to think carefully about the second observation, but it brings up a key point about creating new nodes and adding them to a linked list.

**Program Tip 12.7:** Code that adds a new node to a list must assign a value to some `next` field or the new node will not be linked into the list. Similarly, removing a node from a list also requires an assignment to some `next` field. When recursion is used, the `next` field can be an argument passed recursively. The required assignment to a `next` field can be implemented by a recursive assignment to a parameter that is a reference to a `next` field.

## 12.2.6 Doubly and Circularly Linked Lists, Header Nodes

*Header Nodes.* The special case code for adding a new node to the end of a list we saw in Sect. 12.2.3 can be avoided if we use a **dummy** or **header** node. Using a header node also makes it simpler to remove nodes from a list. A header node ensures that every node in the list has a predecessor. The first node in the linked list is preceded by the header node which isn't considered part of the list.

```
Node * list = 0;           // traditional empty list
Node * header = new Node(); // dummy/header node
```

---

### Program 12.7 listremove.cpp

---

```
void Remove(Node * header, const string& key)
// post: all nodes containing key removed from list/header
{
    Node * before = header;
    Node * list = header->next; // first "real node"

    // invariant: list = before->next, key doesn't appear in header->..->before
    while (list != 0)
    {
        if (list->info == key)
        {
            before->next = list->next; // link around
            delete list;
            list = before->next;      // invariant maintained
        }
        else // invariant maintained
        {
            before = list;
            list = list->next;
        }
    }
}
```

---

listremove.cpp

---

Since the header node is never changed, and all list accesses go through the header, list functions can change the contents in a list without passing the list by reference. You'll need to think carefully about this code to see that it's correct; the invariant should help. Initially no nodes have been examined and the invariant is true. Each time through the loop one of two cases occurs:

- The node being examined, `list`, doesn't contain `key`. In this case both `before` and `list` are advanced.
- We need to remove a node containing `key`. The node before the key-node is linked around the key-node node. We can't advance `before` since the code hasn't examined the node that now comes after `before`.

Writing `Remove` iteratively without a header node is difficult to do correctly. It's much simpler to implement `Remove` recursively. In the following function we assume there is no header node; note that `list` is passed by reference since it changes.

```
void Remove(Node * &list, const string& key)
// post: all nodes containing key removed from list (no header)
{
    if (list != 0)
    {
        Remove(list->next, key);
        if (list->info == key)
        {
            Node * temp = list;
            list = list->next;
            delete temp;
        }
    }
}
```

**Doubly and Circularly Linked Lists.** Linked lists are sequential structures, most operations traverse the list front to back. Some applications require traversal in two directions: from back-to-front as well as front-to-back. For example, a text-editor normally allows the user to move the cursor forward and backward. Implementing a simple editor using a linked list is not difficult if we use **doubly linked list**. In a doubly linked list each node maintains pointers to the node before it in the list as well as to the node after it. This requires one additional data member in the node struct. A diagram of a doubly linked list is shown in Figure 12.8. We'll explore code for manipulating doubly linked lists in the exercises.

In the modified version of Program 12.5, `strlink.cpp`, which we studied in Section 12.2.3, we maintained pointers to both the first and last nodes of a linked list. When both pointers are needed, it's a common convention to use a **circularly linked list**. In a circularly linked list the last node of the list points back to the first node instead of pointing to `NULL`/zero. By keeping only a pointer to the last node of a circularly linked list we can find the first node very simply: `last->next` is the first node. In a circularly linked list with only one node, the last node points to itself since the first node is the last node. The following function counts the nodes in a circularly linked list.

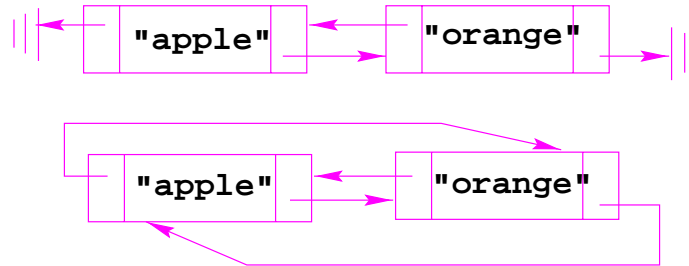


Figure 12.8 Doubly linked list and circular doubly linked list

```
int Count(Node * list)
// pre: list is circularly linked, list points to last node
// post: return # nodes in list
{
    if (list == 0) return 0;    // special case
    int count = 0;             // # nodes
    Node * first = list->next;
    while (first != list)
    {
        count++;
        first = first->next;
    }
    return count + 1;         // count the last node too
}
```

Figure 12.8 shows a doubly linked list that’s also a circularly linked list. The last node points back at the first node, and the first node points at the last node.

Pause to Reflect



**12.22** Write a function `Count` that counts the number of nodes in a linked list. Write the function recursively and with a `while` loop.

**12.23** Write a function `Clone` that returns a copy of its list parameter (assume it’s a linked list of strings.)

```
Node * Clone(Node * list)
// post: return copy of list
```

It’s easiest to write this function recursively, especially if you take advantage of the `Node` constructor.

**12.24** Write a function that returns a pointer to the Node of a linked list that has the minimal value in the list (assume a list of strings and that minimal means first alphabetically.)

```
Node * FindMin(Node * list)
// post: return pointer to minimal node,
//       return 0 if list is empty
```

**12.25** Describe the effects of the function Change that follows.

```
void Change(Node * list)
{
    while (list && list->next)
    {
        Node * temp = list->next;
        list->next = list->next->next;
        list = list->next;
        delete temp;
    }
}
```

**12.26** Describe the effects of the function Chop, where list is a linked list storing int values:

```
void Chop(Node * & list)
{
    if (list != 0)
    {
        Chop(list->next);
        if (list->info % 2 == 0)
        {
            list = list->next;
        }
    }
}
```

**12.27** Write the function CreateList with header as shown. CreateList creates a linked list of  $n$  integers where the first node contains 1 and the last node contains  $n$ . The call Print(CreateList(5)) should print 1 2 3 4 5, where Print is from *strlink.cpp*, Program 12.5.

```
Node * CreateList(int n)
// pre: 0 < n
// post: creates list 1->2->...->n
//       an n node list in which node k contains the int k
```

**12.28** Write the function `GaussList` with header as shown. The function calls `Print(GaussList(4))` should print 1 2 2 3 3 3 4 4 4 4.

```
Node * GaussList(int n)
// pre: 0 < n
// post: returns sorted list, in which
//       k occurs k times, 1 <= k <= n
```

**12.29** Write a function `Reverse` that reverses the order of the nodes in a linked list. Reverse the list by changing pointers, not by swapping `info` fields.

```
void Reverse(Node * & list)
// precondition: list = (a b c ... d)
// postcondition: list = (d ... c b a), list is reversed.
```

**12.30** Write a nonrecursive version of the function `Remove` from Section 12.2.6 where the list doesn't have a header node.

**12.31** Write either an iterative or recursive version of `Remove` that works with doubly linked lists. Assume the list has a header and a tail node where the tail is an extra node at the end of the list serving as sentinel node so that every node has a successor node.

**12.32** Write functions `AddAtFront` and `AddAtBack` that add new nodes to the front and back, respectively, of a circularly linked list.

**12.33** Write a function that doubles a linked list by duplicating each node; that is, the list  $(a b c d)$  is changed to  $(a a b b c c d d)$ . Use the header shown, where `list` is *not* passed by reference. (*Hint*: it's probably easier to write this recursively.)

```
void DoubleList(Node * list)
// precondition: list = (a b c d)
// postcondition: list = (a a b b c c d d)
```

## 12.3 A Templated Class for Sets

To show how linked lists are used in implementing classes we'll develop a class implementing sets similar to `StringSet` (see Section 6.5), but capable of storing elements of any kind, not just strings. Like classes `tvector` and `CList`, the class we design will be templated so that it can represent sets of any type, not just sets of strings. We'll develop a testing program that illustrates how pointers used as instance variables in objects do not always behave as expected.

**Program Tip 12.8:** When developing a templated class, develop a nontemplated version of the class first. Test and debug the nontemplated version first, then implement the templated class. Develop, test, and debug simple programs whose inevitable errors may be easier to find than those in more complex programs.

A set class based on linked lists will not be very efficient, but will eventually lead to a very efficient class when you study another kind of linked structure called a tree.

### 12.3.1 Sets of Strings With Linked Lists

We'll implement a class for representing sets of strings, test and debug the class, then use the tested class as the basis for a templated set class. We'll implement the same methods used in the class `StringSet` (see Program G.7 in How to G) which makes the analysis phase of development simple.



Since we know that searching in a linked list of  $N$  elements is an  $O(N)$  operation, most of the set functions will be  $O(N)$  since they require determining if a string is in the set. The functions we'll implement, their descriptions, and their complexities are given in Table 12.1.

**Table 12.1** Operations for sets of strings implemented using linked lists. Complexities are for a set with  $O(N)$  elements.

operation	description	complexity
construct	make an empty set	$O(1)$
insert	add $s$ to set	$O(N)$
erase	remove $s$ from set	$O(N)$
clear	make set empty	$O(1)$
contains	search for $s$ in set	$O(N)$
size	number of elements in set	$O(1)$

All the  $O(N)$  operations require searching for an element in the set. For example, we'll add a new element at the front of a linked list which is a **constant time** or  $O(1)$  operation. However, we must first determine that the element is not already in the set before adding it. The expression  $O(1)$  is used for an operation whose complexity does not depend on the size of the problem being measured, in this case on the number of elements in the set. Our `clear` function will actually be  $O(N)$ , but we'll explore a constant time version in the exercises.

We'll create a singly linked list with a header node and add new nodes to the front of the list. We'll use the same declarations for member function found in `stringset.h`, Program G.7. Keeping in mind the advice from Programming Tip 9.5, we'll implement a constructor, a method to add elements to a set, and a method for printing the contents of a set. Eventually we'll want to implement an associated iterator class, but at first we'll simply write a set to `cout`, the standard output stream. Our first cut is shown below.

```

class LinkStringSet
{
public:
    LinkStringSet();
    int size() const;
    void print()const;
    void insert(const string& s);
private:
    struct Node
    {
        string info;
        Node * next;
        Node(const string& s, Node * link)
            : info(s), next(link)
        { }
    };
    Node * myFirst; // header node
    int mySize; // # elements in set
};

```

We've already developed code for inserting an element at the front of a linked list and for printing a linked list. We'll need to search for a string before inserting it, but sequential search in a list is nearly identical to sequential search in a vector, so we don't anticipate any difficulties. We'll implement all these methods, test them, then turn to implementing other methods. We won't show the complete test program for these functions, but after testing them thoroughly we can add new methods, knowing any bugs will be in the new methods or in the interactions between the new methods and the already debugged methods.

### 12.3.2 Searching, Clearing, Helper Functions

We've already written the code for `LinkStringSet::contains` since we searched the linked list before inserting a new node at the front. We'd like to reuse this code since we know duplicating code will inevitably lead to a maintenance headache (see Program Tip 4.1). Thinking ahead to the member function `erase` we see that we'll need to search the list to implement that function as well. We'd like to write a private, helper function (see Program Tip 7.14) but what should be the interface of the helper function?

- `contains` returns a boolean value; we need to know if the element is in the set.
- `insert` can use `contains` directly; the location of the element isn't needed since we're adding a new node to the front.
- `erase` removes a node; we need a pointer to the node before the removed node to erase and link around the removed node.

If we used a doubly linked list, the private searching function could return a pointer to the node containing the string being searched for. To remove a node from a singly linked list a pointer to the node being removed won't help; we need a pointer to the node *before*

the node being removed to unsplice the removed node and link around it. We'll write a helper function `findNode` as follows.

```
Node * LinkStringSet::findNode(const string& s) const
// post: returns pointer to node before s
//       returns NULL/0 if !contains(s)
```

We can use `findNode` to implement `contains` and `insert` very easily. Recall that `myFirst` points to a header node so a new node is added *after* the header node.

```
bool LinkStringSet::contains(const string& s) const
// post: return true iff s in set
{
    return findNode(s) != 0;
}
void LinkStringSet::insert(const string& s)
// post: if ! contains(s) then s is added to set
{
    if (! contains(s))
    {
        myFirst->next = new Node(s,myFirst->next);
        mySize++;
    }
}
```

We'll leave the implementation of `findNode` as an exercise, but the header we use above failed to compile when we use it *linkstringset.cpp*. Visual C++ generates the following error message.

```
linkstringset.cpp(56):error C2501:
'Node':missing decl-specifiers (more errors here)
```

The problem is that the declaration `Node` is only known within the `LinkStringSet` declaration. We can use `Node` in parameter lists of member functions, and as the type for a local variable in member function, because member functions “know” about all the class declarations including `Node`. However, the return type of a function is not part of the function's prototype (see the explanation on function overloading in Section 11.1.1) so we must qualify `Node` as follows.

```
LinkStringSet::Node *
LinkStringSet::findNode(const string& s) const
// post: returns pointer to node before s
//       return NULL/0 if !contains(s)
```

If we use this helper function to implement `contains`, and call `contains` from `insert`, we'll need to test `insert` again since its implementation has changed. Once we've tested these functions we'll implement `erase`.

```

void LinkStringSet::erase(const string& s)
// post: ! contains(s), s is removed from set
{
    Node * temp = findNode(s);
    if (temp != 0)
    {
        Node * removal = temp->next; // remove this node
        temp->next = removal->next; // link around
        delete removal;           // delete
        mySize--;
    }
}

```

After testing all the member functions we'll turn to the problem of designing, implementing, and testing an associated iterator class. We'll see that the iterator class methods are very simple to implement, but we'll need to have the iterator access the linked list that's used to implement the `LinkStringSet` class.

### 12.3.3 Iterators and Friend Functions

We'd like to implement a class `LinkStringSetIterator` to access set elements one at a time. The version of `Print` below shows the class used just like any of the iterator classes we've studied so far since all the iterators we develop conform to the same interface. We could easily make this a templated function to work with any iterator as discussed in Section 11.2.2, but we're concerned here with iterating over a `LinkStringSet`.

```

void Print(const LinkStringSet& set)
{
    LinkStringSetIterator it(set);
    for(it.Init(); it.HasMore(); it.Next())
    {
        cout << it.Current() << endl;
    }
    cout << "size = " << set.size() << endl;
}

```

To access elements one at a time we have two choices.

- Provide methods in the class `LinkStringSet` for accessing individual set elements, (i.e., strings stored in the underlying linked list).
- Permit the associated iterator class to access the linked list, but not allow client code to access individual elements.

We'll adopt the second of these options. In general, a **container** class is a **collection** of elements and should have an associated iterator class; the container class provides access exclusively via the associated iterator. The container class and its iterator are tightly coupled (see Program Tip. 6.8) and the iterator class will need to access the

private instance variables of the container class. Access to a class private section can be granted by the class by declaring another class to be a **friend**. The class `Foo` grants

#### Syntax: Declaring friends

```
class Foo
{
    public:
        friend class FooFriend;
    private:
};
```

friend status to the class `FooFriend` whose methods can access private data and helper functions of a `Foo` object. A declaration of friend status is made by the class whose private data will be accessed. It's not possible for a class to request friend status, only for a class to grant friend status. In the iterator class that follows, all the methods are

implemented inline within the class declaration to make it simpler to read the code.

#### Program 12.8 linkstringsetiterator.h

```
class LinkStringSetIterator
{
public:
    LinkStringSetIterator(const LinkStringSet& lset)
        : mySet(lset), myCurrent(0)
    { }
    void Init()
    { myCurrent = mySet.myFirst->next; // first node
    }
    bool HasMore() const
    { return myCurrent != 0;
    }
    string Current() const
    { return myCurrent->info;
    }
    void Next()
    { myCurrent = myCurrent->next;
    }
private:
    typedef LinkStringSet::Node Node;
    const LinkStringSet& mySet;
    Node * myCurrent;
};
```

linkstringsetiterator.h



Each function consists of a single statement that is part of a typical linked list traversal, (e.g., initialization, test, update, and process-element). An iterator is bound to a particular set when the iterator is constructed. As shown, the set is stored as a reference instance variable. We use a `const` reference so that we can iterate over constant sets, for example, in the function `Print` we showed above to demonstrate the `LinkStringSetIterator` class. More information on `const`-ness and iterators is found in [How to D](#).

### 12.3.4 Interactive Testing

We now have both a set class and a friend iterator class. To test the classes we'll use an **interactive** testing program. The program is interactive because the user is given a menu of choices and each choice tests one of the `LinkStringSet` member functions or uses an iterator. The interactive nature allows us to test different cases that we anticipate might cause problems. These include the following.

- Adding the same element more than once
- Deleting an element more than once
- Deleting an element not in the set
- Deleting all the elements then adding new elements
- Clearing the set, adding elements, clearing again

The interactive program can stress the relationships between the member functions, but it's not designed to insert thousands of elements. Stressing the class with large input sets is best done with an **automatic** test program. We'll use the interactive test program *testlinkset.cpp*, Program 12.9. In a larger program, we would use one function for each test case rather than incorporate the code within the `switch` statement. In other words, we would replace

```
case 'i' :  
    word = PromptLnString("enter word : ");  
    set.insert(word);  
    break;
```

With a function call that handles the set insertion.

```
case 'i':  
    DoInsert(set);  
    break;
```

The interactive test program shown here stresses only one set. After we've verified that the set member functions work as expected, or after finding bugs in the functions and fixing them, we'll need to develop a program that uses more than one set to see if problems arise when more than one set is used in the same program. Testing one class is a difficult, time-consuming, but necessary process. Testing a larger program with interacting classes is made simpler if each class is tested separately so that any bugs found are more likely to be from the class interactions rather than from bugs within a class.

**Program Tip 12.9: Every class you develop should be developed with a test suite of programs.** You may want to include both automatic and interactive programs in the test suite. More complex programs with interacting classes will be developed with fewer errors if each individual class is tested separately.

---

Program 12.9 testlinkset.cpp

```
#include <iostream>
#include <string>
#include <cctype>    // for tolower
using namespace std;

#include "linkstringset.h"
#include "prompt.h"

void Print(const LinkStringSet& set)
{
    LinkStringSetIterator it(set);
    cout << "----" << endl;
    for(it.Init(); it.HasMore(); it.Next())
    {   cout << it.Current() << endl;
    }
    cout << "---- size = " << set.size() << endl;
}

void Help()
{
    cout << "(h)elp      print help"      << endl;
    cout << "(i)insert  word into set"    << endl;
    cout << "(c)lear   set"                << endl;
    cout << "(e)rase   word from set"     << endl;
    cout << "(p)rint   the set and size"  << endl;
    cout << "(s)earch  for word in set"  << endl;
    cout << "(q)uit    program"          << endl;
    cout << "-" << endl;
}

void TestSet()
{
    string word, commandLine;
    LinkStringSet set;
    char command = 'h';
    while (command != 'q')
    {   commandLine = PromptLnString("enter command : ");
        if (commandLine == "")
        {   command = 'h';
        }
        else
        {   command = tolower(commandLine[0]);
        }
        switch (command)
        {
            case 'h' :
                Help();
                break;
            case 'i' :
                word = PromptLnString("enter word : ");

```

618

Chapter 12 Dynamic Data, Lists, and Class Templates

```
        set.insert(word);
        break;
    case 'c':
        set.clear();
        break;
    case 'e':
        word = PromptLnString("enter word : ");
        set.erase(word);
        break;
    case 'p':
        Print(set);
        break;
    case 's':
        word = PromptLnString("enter word : ");
        if (set.contains(word))
        {   cout << word << " was found" << endl;
        }
        else
        {   cout << word << " was NOT found" << endl;
        }
    case 'q':
        break;
    default:
        cout << "unrecognized command" << endl;
        break;
    }
}

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

testlinkset.cpp

## OUTPUT

```
prompt> testlinkset
enter command : h
(h)elp      print help
(i)insert   word into set
(c)lear     set
(e)rase     word from set
(p)rint     the set and size
(s)earch    for word in set
(q)uit      program
---
enter command : i
enter word : apple
enter command : i
enter word : cherry
enter command : p
-----
cherry
apple
----- size = 2
enter command : i
enter word : apple
enter command : i
enter word : watermelon
enter command : p
-----
watermelon
cherry
apple
----- size = 3
enter command : s
enter word : cherry
cherry was found
enter command : s
enter word : grapefruit
grapefruit was NOT found
enter command : e
enter word : apple
enter command : p
-----
watermelon
cherry
----- size = 2
```

*output continued*

```

                                     O U T P U T
enter command : c
enter command : p
-----
----- size = 0
enter command : i
enter word : cherry
enter command : p
-----
cherry
----- size = 1
enter command : q

```

### 12.3.5 Deep Copy, Assignment, and Destruction

After thorough testing with an interactive test program we turn to testing more than one class in the same program. Since we're developing a set class we might think about operations we'd like to have that aren't available in the simple `StringSet` class we used as the original model for this class. Typical set operations include union, intersection, and set difference. Before turning to these operations we find a serious flaw in the implementation revealed by the simple Program 12.10, *linksetdemo.cpp*. The program shows that assigning one set to another results in what at first is unexpected behavior. However, thinking back to Program 12.1, *pointerdemo.cpp*, the first program we studied that uses pointers, the results make sense.

---

Program 12.10 *linksetdemo.cpp*

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

// demo of string sets implemented with linked lists

void Print(const LinkStringSet& set)
{
    LinkStringSetIterator it(set);
    for(it.Init(); it.HasMore(); it.Next())
    {
        cout << "\t" << it.Current() << endl;
    }
    cout << "----- size = " << set.size() << endl;
}

```

```

int main()
{
    LinkStringSet a,b;

    a.insert("apple");
    a.insert("cherry");
    cout << "a : "; Print(a);
    b = a;
    cout << "b : "; Print(b);
    a.clear();
    cout << "a : "; Print(a);
    cout << "b : "; Print(b);
    return 0;
}

```

linksetdemo.cpp

## O U T P U T

```

prompt> linksetdemo
a :      cherry
        apple
----- size = 2
b :      cherry
        apple
----- size = 2
a : ----- size = 0
b : ----- size = 2

```

The first printed output for sets `a` and `b` is what we expect. However, after set `a` is cleared, there is nothing in set `b` either, although its size is still two. The problem is that executing the statement `b = a` results in copying the value of the pointer `a.myFirst` to `b.myFirst`. The value of the instance variable `mySize` is copied too, but that doesn't cause a problem. Each set has its own pointer, but both pointers reference the same linked list as shown in Figure 12.9.

Since assignment of one class object to another simply copies the values of each instance variable, the pointers are copied, but the linked lists they point to are not copied. The call `a.clear()` removes all the nodes from `a`'s linked list, which are also the nodes in `b`'s linked list. There's nothing in the set `b`, though the value of `b.mySize` is still two since it's not changed by calling `a.clear()`. When an instance variable points to an object, we may want to copy the object pointed to, not just the pointer, when assigning the class containing the pointer. Copying the object pointed to, and all the objects it may point to, is called a **deep copy**. The default assignment in C++ simply copies pointers, not objects, which is called a **shallow copy**. Before we used pointers we didn't need to worry about these differences because every class we've used behaves properly. Classes that require deep copies, like the `tvector` class, implement the required deep copy

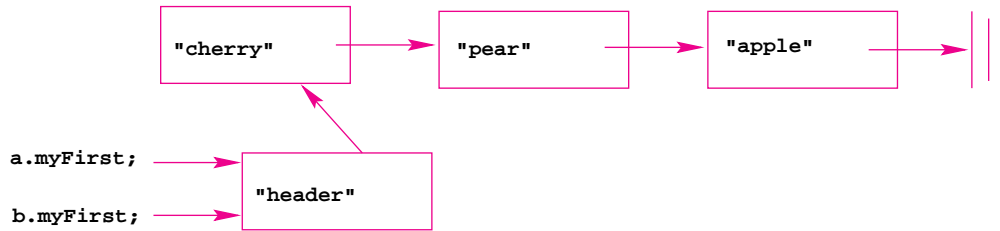


Figure 12.9 Assignment without copying, shared lists.

functions. There are three member functions that must be implemented to generate a deep copy properly: the **copy constructor**, the **assignment operator**, and the **destructor**.

When you design a class, you should aim for the behavior of the class to meet user expectations. For classes like `LinkStringSet` and `tvector` this means that users should be able to assign objects to each other and pass parameters by value if necessary, since the built-in types support these operations. We didn't need to worry about deep copies and shallow copies with the `CList` class because there are no operations that change a `CList` object. Shared storage is only a problem when what's stored changes.

*The Copy Constructor.* The copy constructor is a special constructor called when an object is first defined and initialized from another object of the same type. For example, consider defining several date objects.

```

Date today;
Date tomorrow(today+1); // calls copy constructor
Date yesterday(today-1); // calls copy constructor

yesterday = tomorrow; // calls assignment operator
Date weekago = today-7; // calls copy constructor
  
```

The objects `tomorrow`, `yesterday`, and `weekago` are each constructed and initialized from another `Date` object. The assignment `yesterday = tomorrow` doesn't call a copy constructor because the variable `yesterday` has already been defined. The class copy constructor is called only when an object is first defined, not when it's assigned to or reinitialized in some other way.

Whenever an object is constructed from another object of the same type, a copy constructor is used for the construction and initialization. If you examine the `Date` class you won't see a special constructor, because none is needed.

Every class has a **default copy constructor** that simply copies the value of each instance variable from one object to another. If a shallow copy is acceptable, the default copy constructor is sufficient. Since shallow copies are fine except when there is shared storage, we only need to worry about a copy constructor when there's a shared resource like an object pointed to by an instance variable. Normally only the copy constructor from a `const` object is needed (the top one in the syntax diagram). On rare occasions the behavior of copying from a nonconst object is different and both copy constructors are required.

**Syntax: Copy Constructor**

```
Foo::Foo(const Foo& f);
Foo::Foo(Foo& f);
```

To copy a `LinkStringSet` object we must initialize both instance variables `myFirst` and `mySize`. In the copy constructor that follows, a header node is created in the initializer list. The next field of the header node points to a copy of the linked list that stores the elements of the parameter `set`. The copy is created by the private helper function `clone`. Since a copy of a list will be needed in both the assignment operator and the copy constructor, the code to create the copy is factored out into a helper function.

```
LinkStringSet::LinkStringSet(const LinkStringSet& set)
: myFirst(new Node("header", set.clone())),
  mySize(set.size())
{
    //initializer list makes deep copy
}
```

If you think carefully about the list copy, you'll realize that the private function `clone` is being called by a different object than is making the clone. Private variables and functions can be accessed by any object of the same class.

**The Assignment Operator.** The assignment operator is similar to the copy constructor in making a deep copy, but the assignment operator is called to reinitialize an object that has already been constructed. Because the object being assigned to already exists, some extra bookkeeping is required that wasn't necessary in the copy constructor.

After the assignment `b = a`, `b` will represent a different set than it did before the assignment. The nodes that were part of the old value of `b` should be reclaimed, (e.g., returned to the free store). Two additional requirements should be met by every implementation of an assignment operator. Assignments can be chained together, (e.g.,

**Syntax: Assignment Operator**

```
const Foo&
Foo::operator = (const Foo& f);
```

`a = b = c`), so the assignment operator must return a value. Since assignment is right associative, (e.g., `a = (b = c)`;) the value of the object after assignment is returned. Users may inadvertently write `a = a`. This can cause problems if not checked, so self-assignment should be explicitly guarded in each assignment operator implementation.

The return statement of every assignment operator should be

```
return *this;
```

since an object returns itself after assignment. We don't want to return a copy, we want to return the object itself, so the return type should be a reference, such as `Foo&`. Finally, the reference should be `const` to avoid allowing code like `(a = b).clear()` to compile:

```
const LinkStringSet&
LinkStringSet::operator = (const LinkStringSet& set)
{
    if (this != &set)
    {
        reclaimNodes(myFirst->next);
        myFirst->next = set.clone();
        mySize = set.size();
    }
    return *this;
}
```

To protect against self-assignment, an object checks that the object being assigned to, itself, is different from the object being assigned, `set` in the operator above. We check addresses because we want to guard against assigning the same object, not objects with the same value.

```
string s = "hello";
string t = "hello";
s = t;    // this is fine
s = s;    // guard against weird behavior
```

**The Destructor** A local variable defined in a function is not accessible outside the function. The variable is constructed when the function begins execution, and may accumulate resources as the function executes. Ideally the resources will be reclaimed when they're not needed, which happens when the function returns in the case of a local variable. Consider the variable `set` in the following code.

```
int CountUnique(istream& input)
//post: return # unique words in input
{
    string word;
    LinkStringSet set;
    while (input >> word)
    {
        set.insert(word);
    }
    return set.size();
}
```

Function `CountUnique` correctly counts and returns the number of different or unique words in the stream `input`. What happens to the nodes allocated by `set` after the function returns the size? Although `set` is no longer accessible after `CountUnique` returns, the linked list referenced by `set.myFirst` just before the function returns

will continue to exist after the function returns because the nodes are allocated from the heap; their lifetime is the duration of the program unless the nodes are explicitly deleted.

The destructor member function is called automatically when an object goes out of scope, (e.g., for the local variable `set` when the function above returns). The destructor should take care of reclaiming any resource, particularly storage allocated by `new`.

#### Syntax: Class Destructor

```
Foo::~~Foo();
```

The destructor has the same name as the class it belongs to, but is preceded by a tilde: `~`.<sup>4</sup> When you first implement a class, the destructor should

be a stub function. After you've debugged other member functions, implement the destructor to reclaim storage (or other resources). The advice in Program Tip 12.3 makes particular sense when you're implementing a destructor.

```
LinkStringSet::~~LinkStringSet()
{
    reclaimNodes(myFirst);
    myFirst = 0;
}
```

We can call the helper function `reclaimNodes` that we used in the assignment operator. Since nodes are reclaimed in both places it makes sense to factor out the code into a helper function. In the implementation of `LinkStringSet` we would make `reclaimNodes` a stub function and implement it after debugging other member functions.

**Program Tip 12.10:** When you implement one of the following three member functions, it is normally an indication that you should implement all three functions.

1. Copy constructor, for initializing an object based on another object of the same type.
2. Assignment operator `=`, for assigning a new value of the same type to an existing object.
3. Destructor, for reclaiming resources allocated by an object during its lifetime, (e.g., memory allocated by `new`).

### 12.3.6 A Templated Version of `LinkStringSet`

With the string set class debugged, we'll turn to creating a templated version of the class. We'll call the new class `LinkSet` and we'll define variables of type `LinkSet<string>` and `LinkSet<int>` among the many kinds of sets we can create. Most of the changes in creating a templated class are syntactic in nature. I copied the header file `linkstringset.h`

<sup>4</sup>The tilde `~` is sometimes pronounced "twiddle," but tilde is an acceptable pronunciation.

(accessible with the code that comes with this book) to the file *linkset.h*. I automatically replaced every occurrence of `string` with `T`, the identifier I used for the template parameter. I replaced all occurrences of `LinkStringSet` with `LinkSet` too. To indicate the class is templated, I added the following line whose syntax is the same as the declaration for creating a templated function as shown, for example, in Section 11.2.

```
template <class T>
```

The only other changes needed in the header file were for the iterator class. The name had been changed to `LinkSetIterator` when I changed all occurrences of `LinkStringSet` to `LinkSet`. I added the same template declaration before the class that I used to indicate that `LinkSet` was a templated class. Finally, I changed the friend declaration in `LinkSet` as follows to indicate that the iterator class is templated.

```
friend class LinkSetIterator<T>;
```

The compiler needs information that `LinkSetIterator` is a templated class to parse this friend declaration, so I added the following forward declaration just before the class `LinkSet` (see Program Tip 12.2 for reasons to use forward references).

```
template <class T> class LinkSetIterator;
```

In the iterator class declaration, all occurrences of `LinkSet` must be replaced with `LinkSet<T>` to indicate that the class `LinkSet` is templated. This yields the complete declaration *linkset.h*.

---

#### Program 12.11 linkset.h

```
#ifndef _LINKSET_H
#define _LINKSET_H

template <class T> class LinkSetIterator;

template <class T>
class LinkSet
{
public:
    LinkSet();

    // methods for deep copy
    LinkSet(const LinkSet& set);
    const LinkSet& operator =(const LinkSet& set);
    ~LinkSet();

    // accessors
    bool contains(const T& s) const; // true iff s in set
    int size() const; // # elements in set

    // mutators
    void insert(const T& s); // add to set
    void erase(const T& s); // remove from set
};
```

```

void clear();           // delete all elements

friend class LinkSetIterator<T>;

private:

struct Node
{
    T info;
    Node * next;
    Node(const T& s, Node * link)
        : info(s), next(link)
    { }
};

Node * findNode(const T& s) const; // helper
void  reclaimNodes(Node * ptr);    // delete/reclaim
Node * clone() const;             // copy list

Node * myFirst;
int   mySize;
};

template <class T>
class LinkSetIterator
{
public:
    LinkSetIterator(const LinkSet<T>& lset)
        : mySet(lset),
          myCurrent(0)
    { }

    void Init()
    {
        myCurrent = mySet.myFirst->next; // first node
    }

    bool HasMore() const
    {
        return myCurrent != 0;
    }

    T Current() const
    {
        return myCurrent->info;
    }

    void Next()
    {
        myCurrent = myCurrent->next;
    }

private:
    typedef LinkSet<T>::Node Node;
    const LinkSet<T>& mySet;
    Node * myCurrent;
};

#include "linkset.cpp"

#endif

```

linkset.h

Notice that the last line of the header file (before the #endif) is an include directive:

```
#include "linkset.cpp"
```

Templated classes, like templated functions, are used to instantiate class code rather than being class code (see Section 11.2.3.) When client code instantiates a templated class by defining objects, the template class declarations are used to generate code for the specific type used in the instantiation.

```
ListSet<string> sset;
ListSet<int>    iset;
ListSet<Date>  dset;
ListSet<int>   iset2;
```

The four set definitions here generate code for three different `ListSet` instantiations: one for `int` sets, one for `Date` sets, and one for `string` sets. The compiler is smart enough to instantiate the `int` set code only once even though two objects are defined — only the first instantiation of a templated class actually creates code.

The compiler must be able to find definitions for the member functions of a templated class *when the class is instantiated*. This is a different process than is used for nontemplated classes. When we create nontemplated class definitions, such as, as in *linkstringset.cpp* or *date.cpp*, the definitions can be compiled into object code that is linked with client code to create an executable. It's not possible to compile the definitions in *linkset.cpp* because these definitions are not code, they're used to generate code when a `ListSet` is instantiated. Because client programs typically include `.h` files that specify interfaces, a templated-class interface file usually includes the corresponding implementation or `.cpp` file as it does in *linkset.h*, Program 12.11. The compiler then has access to the template definitions so that they can be compiled into object code when they're instantiated by the client program.

**Program Tip 12.11: The compiler must access both interface and implementation when instantiating a templated class. Typically templated classes are defined inline, within the class declaration, or separately in a `.cpp` file that is included by the corresponding `.h` file.** In either case the compiler has access to the template definitions when client code instantiates a templated class. The C++ standard specifies that only those member functions that are called by a client program are instantiated.

If a client program that uses `ListSet<int>` objects calls only `insert` and `size`, but never `contains`, `clear`, or `erase`, then code for the functions not-called in the client program will *not* be instantiated by the compiler. The compiler tries to minimize the code created so that the programmer is freed from that worry.

**The `LinkSet` Implementation: *linkset.cpp*.** The code in *linkset.cpp*, Program 12.12, highlights the massive syntactic ugliness of template-class member function definitions. When you first read these definitions, try to ignore the `template <class T>` that precedes each method definition. This is the same syntax for declaring templated functions we saw in Section 11.2, but reproduced once for each method. Since `LinkSet` is

a templated class, the class name that qualifies each method must somehow indicate the template parameter. Instead of writing

```
int LinkSet::size() const
```

we must write

```
template <class T>
int LinkSet<T>::size() const
```

to indicate that the definition is for the class `LinkSet` templated on a type argument `T`.

I created *linkset.cpp* by copying the implementation file *linkstringset.cpp*. I first replaced every occurrence of `LinkStringSet` with `LinkSet<T>`.<sup>5</sup> I then replaced every occurrence of `string` with `T`. Finally, I added `template <class T>` before each member function.

---

#### Program 12.12 linkset.cpp

---

```
#include "linkset.h"

template <class T>
LinkSet<T>::LinkSet()
    : myFirst(new Node(T(),0)),
      mySize(0)
{
    // header node created
}

template <class T>
bool LinkSet<T>::contains(const T& s) const
{
    Node * temp = findNode(s);
    return temp != 0;
}

template <class T>
int LinkSet<T>::size() const
{
    return mySize;
}

template <class T>
void LinkSet<T>::insert(const T& s)
{
    if (! contains(s))
    {
        myFirst->next = new Node(s,myFirst->next);
        mySize++;
    }
}

template <class T>
void LinkSet<T>::erase(const T& s)
```

---

<sup>5</sup>This caused two problems with constructors since `LinkSet<T>::LinkSet()` is the default constructor, not `LinkSet<T>::LinkSet<T>()`; and a similar problem with the destructor name.

630

## Chapter 12 Dynamic Data, Lists, and Class Templates

```

{
    Node * temp = findNode(s);
    if (temp != 0)
    {
        Node * removal = temp->next;
        temp->next = removal->next;
        delete removal;    // can we reuse this?
        mySize--;
    }
}

template <class T>
void LinkSet<T>::reclaimNodes(Node * ptr)
{
    if (ptr != 0)
    {
        reclaimNodes(ptr->next);
        delete ptr;
    }
}

template <class T>
void LinkSet<T>::clear()
{
    reclaimNodes(myFirst->next);
    myFirst->next = 0;    // nothing in the set
    mySize = 0;
}

template <class T>
LinkSet<T>::Node * LinkSet<T>::findNode(const T& s) const
// post: returns pointer to node before s or NULL/0 if !contains(s)
{
    Node * list = myFirst; // list non-zero

    while (list->next != 0 && list->next->info != s)
    {
        list = list->next;
    }
    if (list->next == 0) return 0;
    return list;
}

template <class T>
LinkSet<T>::LinkSet(const LinkSet<T>& set)
    : myFirst(new Node(T(),set.clone())),
      mySize(set.size())
{
    // initializer list made deep copy
}

template <class T> const LinkSet<T>&
LinkSet<T>::operator = (const LinkSet<T>& set)
{
    if (this != &set)
    {
        reclaimNodes(myFirst->next);
        myFirst->next = set.clone();
        mySize = set.size();
    }
    return *this;
}

```

```

}

template <class T>
LinkSet<T>::~~LinkSet()
{
    reclaimNodes(myFirst);
    myFirst = 0;
}
template <class T>
LinkSet<T>::Node * LinkSet<T>::clone() const
{
    Node front(T(),0); // node, not pointer, anchors copy
    Node * last = &front; // be wary of using address of operator!

    Node * temp = myFirst->next;
    while (temp != 0)
    {
        last->next = new Node(temp->info,0);
        last = last->next;
        temp = temp->next;
    }
    return front.next;
}

```

linkset.cpp

When I first tested the templated class, I created `LinkSet<string>` objects and used the same testing programs that helped test the original nontemplated `LinkStringSet` class. Then I added `LinkSet<int>` definitions and discovered two small problems that were simple to fix. The constructor definition for the nontemplated class `LinkStringSet` follows.

```

LinkStringSet::LinkStringSet()
: myFirst(new Node("header",0)),
  mySize(0)
{
    // header node created
}

```

Using the cut-paste-and-change technique for creating a templated version generated this constructor.

```

template <class T>
LinkSet<T>::LinkSet()
: myFirst(new Node("header",0)),
  mySize(0)
{
    // header node created
}

```

This definition works fine with a string set, but fails with an int set. Can you see why? The problem is in the construction of the header node. The private struct `Node` is now templated, so it cannot be initialized with a string. Instead, we use the default

constructor for the template type `T`, written as `T()` as shown in each `Node` construction in `linkset.cpp`, Program 12.12.

Following the advice outlined in Program Tip 12.8 made it very easy to create the templated class once the nontemplated class had been designed, implemented, debugged, and tested. Because the syntax of templated classes is daunting at first, following this advice is a good idea. It remains a good idea even after you have considerable experience programming using C++.

## Pause to Reflect



- 12.34** In the implementation of `linkset.cpp`, Program 12.12, the function `clone` is a `const` function. Is this necessary? Is the function called somewhere on a `const` set?
- 12.35** Why is the declaration of `Node` in the set classes in the private section and not in the public section?
- 12.36** Why is the `LinkSet<T>::clear()` function  $O(N)$  as implemented? Can you think of a modification to the class that results in a constant time  $O(1)$  implementation of `clear`? (Hint: put off deletion as long as possible.)
- 12.37** Suppose you're forming the union of two `LinkStringSet` objects `a` and `b`. The union is a new set containing all the elements in both `a` and `b`. If `a` has 10 elements and `b` has 100 elements, does the order in which elements from the sets are inserted into the new set being constructed make a difference (i.e., should elements from the small set be inserted before elements from the big set, or *vice versa*)?
- 12.38** If sets are implemented using a sorted vector instead of a linked list so that `contains` is an  $O(\log N)$  operation using binary search, does the order in which the union of two sets is done (see the previous question) make more of a difference? Why?
- 12.39** The assignment operator returns a reference to the object just assigned to. If the return type is a copy instead of a reference, (e.g., `LinkStringSet` instead of `const LinkStringSet&`), the copy constructor must be called to create the copy. Why is a copy less than ideal?
- 12.40** In the final version of `clone` in `linkset.cpp`, Program 12.12, a local `Node` named `front` is defined, and the address of `front` assigned to `last`. What's the purpose of the assignment and definition and what's an alternative that avoids using `&`, the address-of operator.

## 12.4 Chapter Review

In this chapter we discussed pointers. Pointers are indirect references, useful when data need to be accessed in more than one way and when data must be allocated dynamically. We discussed sharing objects between classes using reference variables and pointers. We

also discussed self-referential data structures called linked lists that have many applications. It's possible to insert new elements into a linked list without shifting the existing elements, making linked lists the method of choice for many sparse structures. We studied copy constructors, assignment operators, and destructors, three member functions often required when instance variables point to objects on the heap. We also saw an example of designing, implementing, and testing a templated class by starting with a nontemplated class.

Topics covered include:

- Variables have names, values, and addresses. The address of a variable can be assigned to a pointer.
- As part of defensive programming, make pointers point to objects allocated on the heap using `new`, not to objects allocated on the stack.
- Several operators are used to manipulate pointers: `->`, `*`, `&`, and operators `new`, and `delete`.
- Pointers can be used for efficiency since a `tvector` of pointers to strings requires less space than a `tvector` of strings, especially if the `tvector` is not full.
- The `new` operator is used to allocate memory dynamically from the heap. Memory can be allocated using `new` in conjunction with a constructor with arguments.
- Pointers are dereferenced to find what they point to. Pointers can be assigned values in four ways: using `new`, using `&` to take the address of existing storage (not a good idea, in general), assigning the value of another pointer, and assigning `0` or `NULL`.
- A destructor member function is called automatically when an object goes out of scope. Any memory allocated using `new` during the lifetime of the object should be freed using `delete` in the destructor.
- Reference instance variables can be used to share an object among more than one object. Reference instance variables must be initialized at construction; once constructed and bound to an object, a reference variable cannot be bound to a different object (unlike a pointer, for example.)
- Pointers can be used to change the values of parameters indirectly. This is how parameters are changed in C: addresses are passed rather than values. The indirect addresses are used to change values.
- Linked lists support splicing, or fast insertions and deletions (in contrast to vectors in which items are often shifted during insertion and deletion). However, items near the end of a linked list take more time to access than items near the front.
- Recursive linked list functions (sometimes with pointers passed by reference) are often shorter than an equivalent iterative version of the function.
- A header node can be used when implementing linked lists to avoid lots of special-case code, especially when deleting and inserting elements.
- Doubly and circularly linked lists are alternatives to singly linked lists.
- Classes can be templated so that they can be used to generate literally thousands of different classes, just as templated functions represent thousands of functions.

## 12.5 Exercises

**12.1** Implement quicksort for linked lists. The partition function should divide a list into two sublists, one containing values less than or equal to the pivot, the other containing values greater than the pivot. Conceptually the partition function returns three things:

- The pivot element (a node).
- The list of items less than or equal to the pivot element.
- The list of items greater than or equal to the pivot element.

Since you'll need to join lists together after recursively sorting, you'll need to think carefully about how to develop the program. You might, for example, maintain pointers to the first and last nodes of each list returned from the partition function. Alternatively, you could maintain a pointer to the last node and make these lists circular.

When you've implemented the sort, develop a test program to verify that the original list is sorted. Then time the sort using either randomly constructed large lists or by reading words from a text file and sorting them. Consider writing a templated version of the sort as well.

**12.2** Develop an implementation of merge sort for linked lists. Merge sort is described in the exercises of Chapter 11. Write two functions, one to merge two sorted lists and one to implement the merge sort.

```
Node * merge(Node * lhs, Node * rhs)
// pre: lhs is sorted, rhs is sorted
// post: returns sorted list containing all nodes
//       from lhs and rhs no new nodes are created,
//       nodes are relinked, complexity is O(a + b),
//       where a = # nodes in lhs, b = # nodes in rhs

void mergesort(Node * & list)
// post: list is sorted
//       (re-arranging pointers, not copying values)
```

Write a program to test the sort on linked lists of strings. Then compare the runtime of your sort with the time to copy the values from a list into a vector, sort the vector using the merge sort code from *sortall.h*, then copy the values back into the linked list.

**12.3** The *Josephus problem* (see [Knu98b]) is based on a “fair” method for designating one person from a group of  $N$  people. Assume that the people are arranged in a circle and are numbered from 1 to  $N$ . If we count off every fourth person, removing a person as we count them off, then the first person removed is number 4. The second person removed is number 8, the third person removed is 5 (because the fourth person is no longer in the circle), and so on. Write a program to print the order in which people are removed from the circle given  $N$ , the number of people, and  $M$ , the number used to count off. The problem originates from a group determined to commit suicide rather than surrender or be killed by the enemy. Consider using a doubly linked or a circularly linked list as appropriate.

- 12.4** Write a program to automatically stress/test the class `LinkStringSet` or its templated equivalent `LinkSet`. The program should insert thousands of items, delete thousands, and in general exercise each set method. For each test, develop a rationale for why you've chosen the test as a way of stressing the implementation.

When you've developed the program, change the set implementation in the manner described below and see if the change results in improved running times. You'll need to instrument your test program using `CTimer` objects (see *ctimer.h*, Program G.5, in How to G) to judge if the implementation is more efficient.

The current set implementations "reclaims" nodes by deleting them when one set is assigned to another or when a set object's destructor is called. Instead of deleting nodes, add the reclaimed nodes to a **static class** linked list of free nodes.



```
// in linkset.h
template <class T>
class LinkSet
{
    ...

private:

    static Node * ourFreeList;
};
```

```
// in linkset.cpp
template <class T> LinkSet<T>::Node *
LinkSet<T>::ourFreeList = 0; // initially empty
```

The idea is that there is one linked list shared by all `LinkSet<T>` objects — recall that a static class variable is shared by all objects (see Section 10.4.3). When nodes are reclaimed, they are added to the front of the static, shared linked list. When nodes are needed, (i.e., during insertion), the shared linked list of free nodes is used as a source of nodes before `new` is called. Nodes are allocated using `new` only if there are no nodes on the list pointed to by `ourFreeList`.

Implement this change and time the program to see if it's more efficient to maintain a free list of nodes than to use the system freestore.

- 12.5** In Section 10.5.5 a class for representing polynomials was developed. The class used a `CList` list to store terms. Reimplement the class using linked lists. You'll need to implement a copy constructor, an assignment operator, and a destructor that were not needed in the original implementation of the class `Poly`. Shallow copies were fine in that implementation because it's not possible to change a `CList` object, only to create a new object. The new implementation should create copies of polynomials as needed, but change a polynomial, for example, when operator `+=` is used to add a term to a polynomial.

Test the program and compare its performance to the original implementation. You'll need to develop automated testing functions that stress the polynomial class by creating huge polynomials, adding them, multiplying them, and so on.

- 12.6** Implement free functions for creating the union and intersection of two `LinkSet<T>` objects. The union of two sets is denoted  $a \cup b$ , it is a set containing all the elements

in either  $a$  or  $b$ . The intersection of two sets is denoted  $a \cap b$ ; it is a set containing those elements that are common to both  $a$  and to  $b$ .

```
LinkSet<string> a, b, c, d;
// fill a and b with values

c = union(a,b); // c is the union of a and b
d = intersect(a,b); // d is the intersection of a and b
```

When you've tested these functions, overload operator `+` for union and operator `*` for intersection. This means you should also implement overloaded operators `+=` and `*=` (see the guidelines for overloading operators in Section 9.4 or How to E).

- 12.7** Write a program to implement a *kid/toy* simulation. A file stores information about available toys in a format specific to this problem.

```
wooden blocks      : sturdy
choo-choo train   : sturdy
bucket and shovel : durable
talking doll      : sturdy
hothot wheels     : durable
wickets, mallets, and croquet balls : durable
nose glasses      : flimsy
mr. zucchini head : flimsy
```

There are least three categories of toy: sturdy, durable, and flimsy. These categories indicate how long a toy can be played with before it breaks and must be fixed. Toys don't wear out in this model, they can be fixed many times and last forever. All discussions are in "play units," which can be thought of as hours. A sturdy toy breaks 2% of the time it's played with, a durable toy breaks 15% of the time, and a flimsy toy breaks 45% of the time. We'll interpret this as a probability, so each hour (play unit) a sturdy toy is played with, there's a 2 in 100 chance it will break. Broken toys require time to repair: sturdy toys can be fixed in one hour, durable toys in two hours, and flimsy toys in four hours.

The program should read a data file and construct a *toy chest* from which toys are borrowed to be played with. Kids use toys. The number of kids in a simulation is specified when the simulation begins. During one step of the simulation, each kid takes a turn playing with his/her toy. At the next step, the order in which kids take turns changes; the order should be shuffled using a shuffling function like the one in *shuffle.cpp*, Program 8.4. The number of steps in the simulation is specified when the simulation begins.

When a toy breaks, it must be placed back in the toy chest and remain there until it is fixed. A kid putting a toy in the chest takes a new toy out of the chest. If there are no toys in the chest, the kid picks another kid at random and shares that kid's toy. The toy is shared until it breaks or until one of the kids gets bored with the toy. A toy can be shared among everyone, there's no limit, but each time a kid plays with a toy counts as a "play unit." Kids like flimsy toys, so they get bored less often with flimsy toys than they do with sturdy toys. After playing with a toy, a kid trades the toy in for a new toy (or for sharing someone's toy) if bored. A kid gets bored after playing with the same toy for  $n$  play units/hours, where  $n = 2$  for sturdy toys,  $n = 3$  for durable toys, and  $n = 4$  for flimsy toys.

- 12.8** Boggle is a game of finding words by connecting letters on a two-dimensional grid. Design and implement a program to find all the words on a grid based on structuring data using sets as described in this exercise. The output of some of the words that begin with 'a' found on a randomly generated board is shown below. For each word, a list of the positions in which the letters of the word appear on the board is shown (positions give row and column indexes using matrix coordinates: (0,0) is the upper left corner).

```

                                O U T P U T

prompt> wordgame
board size  between 3 and 8: 7
  g n t b s h z
  d s w u u d r
  e n u a a i a
  z z m e b e a
  u a t y r i y
  i y n e v p a
  d s o s r t o
file of words: gamewords
abet      (2, 4) (3, 4) (3, 3) (4, 2)
aid       (4, 1) (5, 0) (6, 0)
air       (5, 6) (4, 5) (4, 4)
airy      (5, 6) (4, 5) (4, 4) (4, 3)
amaze     (2, 3) (3, 2) (4, 1) (3, 1) (2, 0)
amuse     (4, 1) (3, 2) (2, 2) (1, 1) (2, 0)
more words found ...

```

Letters are considered adjacent if they touch horizontally, vertically, or diagonally (see the output for examples). Once a grid position is used in forming a word, the position cannot be used again in the same word.

There are many ways to find all the words; the method suggested here uses sets and is relatively straightforward to implement, though certainly not trivial. Part of a class `WordGame` declaration is shown as *wordgame.h*

---

#### Program 12.13 *wordgame.h*

```

#ifndef _WORDGAME_H
#define _WORDGAME_H

#include <string>
using namespace std;
#include "point.h"
#include "tvector.h"
#include "linkset.h"

```

```

class WordGame
{
public:
    WordGame(int size); // max grid size
    void MakeBoard(); // create a grid of letters

    // is a word on the board? one version returns locations
    bool OnBoard(const string& s);
    bool OnBoard(const string& s, tvector<Point>& locations);

    // other functions

private:
    typedef LinkSet<Point> PointSet;
    typedef LinkSetIterator<Point> PointSetIterator;

    tmatrix<char> myBoard;
    tvector<PointSet> myLetterLocs;
    PointSet myVisited;

    bool IsAdjacent(const Point& p, const Point& q);
    bool OnBoardAt(const string& s, const Point& p,
                  tvector<Point>& locs);
};

```

wordgame.h

---

The instance variable `myLetterLocs` is the key to the program. It's a vector of 26 sets, each set stores positions (positions are recorded using the struct `Point` from *point.h*, Program G.10). The value of `myLetterLocs[0]` is the set of locations at which the letter d'`a' appears on the board. Similarly, `myLetterLocs[1]` records all locations of the letter 'b', and so on. These sets are initialized when the board is constructed. The private helper function `OnBoardAt` works using **backtracking**, discussed in the exercises from Chapter 11. You must determine how this function works and implement the other functions to find all words in a file of words.

---

#### Program 12.14 wordgame.cpp

```

bool Boggle::OnBoardAt(const string& s, const Point& p, tvector<Point>& locs)
// post: return true iff string s can be found on the board
//       beginning at location p (s[0] found at p, s[1] at a location
//       adjacent to p, and so on). If found, locs stores the locations
//       of the word, locations are added using push_back
{
    if (s.length() == 0) return true; // all letters done, found the word

    PointSet ps = myLetterLocs['z' - s[0]]; // set of eligible letters
    PointSetIterator psi(ps); // try all locations
    for(psi.Init(); psi.HasMore(); psi.Next())
    {
        Point nextp = psi.Current();
        if (IsAdjacent(p,nextp) && ! myVisited.contains(nextp))
        {
            myVisited.insert(nextp);
            locs.push_back(nextp);
            if (OnBoardAt(s.substr(1,s.length()-1),nextp,locs))

```

```

        {   return true;
        }
        locs.pop_back();
        myVisited.erase(nextp);
    }
}
return false; // tried all locations, word not on board
}

```

wordgame.cpp

**12.9** A **stack** is a data structure sometimes called a **LIFO** structure, for “last in, first out.” A stack is modeled by cars pulling into a driveway: the last car in is the first car out. In a stack, only the last element stored in the stack is accessible. Rather than use insert, remove, append, or delete, the vocabulary associated with stack operations is

- **push**—add an item to the stack; the last item added is the only item accessible by the `top` operation.
- **top**—return the topmost, or most recent, item pushed onto the stack; it’s an error to request the top item of an empty stack.
- **pop**—delete the topmost item from the stack; it’s an error to pop an empty stack.

For example, the sequence `push(3)`, `push(4)`, `pop`, `push(7)`, `push(8)` yields the stack (3,7,8) with 8 as the topmost element on the stack.

Stacks are commonly used to implement recursion, since the last function called is the first function that finishes when a chain of recursive clones is called.

Write a (templated) class to implement stacks (or just implement stacks of integers). In addition to member functions `push`, `pop`, and `top`, you should implement `size` (returns number of elements in stack), `clear` (makes a stack empty), and `isEmpty` (determines if the stack is empty). Use either a vector or a linked list to store the values in the stack. Write a test program to test your stack implementation.

After you’ve tested the `Stack` class, use it to evaluate **postfix** expressions. A postfix expression consists of two values followed by an operator. For example: `3 5 +` is equal to 8. However, the values can also be postfix expressions, so the following expression is legal.

```
3 5 + 4 8 * + 6 *
```

This expression can be thought of as parenthesized, where each parenthesized subexpression is a postfix expression.

```
( ( ( 3 5 + ) ( 4 8 * ) + ) 6 * )
```

However, it’s easy to evaluate a postfix expression from left to right by pushing values onto a stack. Whenever an operator (`+`, `*`, etc.) is read, two values are popped from the stack, the operation computed on these values, and the result pushed back onto the stack. A legal postfix expression always leaves one number, the answer, on the stack. Postfix expressions do not require parentheses;  $(6 + 3) \times 2$  is written in postfix as `6 3 + 2 *`. Write a function to read a postfix expression and evaluate it using a stack.