23

23 Intermediate class

The class construct has many ramifications and extensions, a few of which are introduced in this chapter.

"static" class members for shared data

Section 23.1 looks at the problem of data that need to be shared by all instances of a class. Shared data are quite common. For example, the air traffic control program in Chapter 20 had a minimum height for the aircraft defined by a constant; but it might be reasonable to have the minimum height defined by a variable (at certain times of the day, planes might be required to make their approaches to the auto lander somewhat higher say 1000 feet instead of 600 feet). The minimum height would then have to be a variable. Obviously, all the aircraft are subject to the same height restriction and so need to have access to the same variable. The minimum height variable could be made a global; but that doesn't reflect its use. If really is something that belongs to the aircraft and so should somehow belong to class `Aircraft`. C++ classes have "`static`" members; these let programmers define such shared data.

Friends – sneaking through the walls of privacy

Section 23.2 introduces "`friends`". One of the motivations for classes was the need to build privacy walls around data and specialist housekeeping functions. Such walls prevent misuse of data such as can occur with simple structs that are universally accessible. Private data and functions can only be used within the member functions of the class. But sometimes you want to slightly relax the protection. You want private data and functions to be used within member functions, and in addition in a few other functions that are explicitly named. These additional functions may be global functions, or they may be the member functions of some second class. Such functions are nominated as "`friends`" in a class declaration. (The author of a class nominates the friends if any. You can't come along later and try to make some new function a "friend" of an existing class because, obviously, this would totally defeat the security mechanisms.) There aren't many places where you need friend functions. They sometimes appear when you have a cluster of separate classes whose instances need to work together closely. Then you may get situations where there a class has some data members or functions that you would like to make accessible to instances of other members of the class cluster without also making them accessible to general clients.

Iterators

Section 23.3 introduces iterators. Iterator classes are associated with collection classes like those presented in Chapter 21. An Iterator is very likely to be a "friend" of the collection class with which it is associated. Iterators help you organize code where you want to go through a collection looking at each stored item in turn.

Operator functions

My own view is that for the most part "operator functions", the topic of Section 23.4, are an overrated cosmetic change to the ordinary function call syntax. Remember how class `Number` in Chapter 19 had functions like `Multiply()` (so the code had things like `a.Multiply(b)` with `a` and `b` instances of class `Number`)? With operator functions, you can make that `a * b`. Redefining the meaning of operator `*` allows you to pretty up such code.

Such cosmetic uses aren't that important. But there are a few cases where it is useful to redefine operators. For instance, you often want to extend the interface to the iostream library so that you can write code like `Number x; … cout << "x = " << x << endl`. This can be done by defining a new global function involving the `<<` operator. Another special case is the assignment operator, operator `=`; redefinition of operator `=` is explained in the next section on resource manager classes. Other operators that you may need to change are the pointer dereference operator, `->` and the `new` operator. However, the need to redefine the meanings of these operators only occurs in more advanced work, so you wont see examples in this text.

Resource manager classes

Instances of simple classes, like class `Number`, class `Queue`, class `Aircraft` are all represented by a single block of data. But there are classes where the instances own other data structures (or, more generally, other resources such as open files, network connections and so forth). Class `DynamicArray` is an example; it owns that separately allocated array of `void*` pointers. Classes `List` and `BinaryTree` also own resources; after all, they really should be responsible for those listcells and treenodes that they create in the heap.

Destructor functions for resource manager classes

Resource managers have special responsibilities. They should make certain that any resources that they claim get released when no longer required. This requirement necessitates a new kind of function – a "destructor". A destructor is a kind of counterpart for the constructor. A constructor function initializes an object (possibly claiming some resources, though usually additional resources are claimed later in the object's life). A destructor allows an object to tidy up and get rid of resources before it is itself discarded. The C++ compiler arranges for calls to be made to the appropriate destructor function whenever an object gets destroyed. (Dynamic objects are destroyed when you apply operator `delete`; automatic objects are destroyed on exit from function; and static objects are destroyed during "at_exit" processing that takes place after return from `main()`.)

Operator = and resource manager classes

There is another problem with resource manager classes – assignment. The normal meaning of assignment for a struct or class instance is "copy the bytes". Now the bytes in a resource manager will include pointers to managed data structures. If you just copy the bytes, you will get two instances of the resource manager class that both have pointers to the same managed data structure. Assignment causes sharing. This is very rarely what you would want.

If assignment is meaningful for a resource manager, its interpretation is usually "give me a copy just like this existing X"; and that means making copies of any managed resources. The C++ compiler can not identify the managed resources. So if you want assignment to involve copying resources, you have to write a function does this. This becomes the "assignment function" or "`operator=()`" function. You also have to write a special "copy constructor".

Preventing assignment

Actually, you usually want to say "instances of this resource manager class cannot be assigned". Despite examples in text books, there are very few situations in real programs where you want to say something like "give me a binary tree like this existing binary tree". There are mechanisms that allow you to impose constraints that prohibit assignment.

Inheritance

The final section of this chapter introduces the idea of inheritance. Basically, inheritance allows you to define a new class that in some way extends an existing defined class. There are several different uses for inheritance and the implications of inheritance are the main topic of Part V of this text.

Although your program may involve many different kinds of object, there are often similarities among classes. Sometimes, it is possible to exploit such similarities to simplify the overall design of a program. An example like this is used to motivate the use of class hierarchies where specialized classes inherit behaviours from more general abstract classes.

The next subsection shows how class hierarchies can be defined in C++ and explains the meanings of terms like "virtual function". Other subsections provide a brief guide to how programs using class hierarchies actually work and cover some uses of multiple inheritance.

23.1 SHARED CLASS PROPERTIES

A class declaration describes the form of objects of that class, specifying the various data members that are present in each object. Every instance of the class is separate, every instance holds its own unique data.

Sometimes, there are data that you want to have shared by all instance of the class. The introduction section of this chapter gave the example of the aircraft that needed to "share" a minimum height variable. For second example, consider the situation of writing a C++ program that used Unix's Xlib library to display windows on an Xterminal. You would probably implement a class Window. A Window would have data members for records that describe the font to be used for displaying text, an integer number that identifies the "window" actually manipulated by the interpretive code in the Xterminal itself, and other data like background colour and foreground colour. Every Window object would have its own unique data in its data members. But all the windows will be displayed on the same screen of the same display. In Xlib the screen and the display are described by data structures; many of the basic graphics calls require these data structures to be included among the arguments.

You could make the "Display" and the "Screen" global data structures. Then all the Window objects could use these shared globals.

But the "Display" and the "Screen" should only be used by Windows. If you make them globals, they can be seen from and maybe get misused in other parts of the program.

The C++ solution is to specify that such quasi globals be changed to "class members" subject to the normal security mechanisms provided by C++ classes. If the variable that represents the minimum height for aircraft, or those that represent the Display and Screen used by Windows, are made private to the appropriate classes, then they can only be accessed from the member functions of those classes.

Of course, you must distinguish these shared variables from those where each class instance has its own copy. This is done using the keyword `static`. (This is an unfortunate choice of name because it is a quite different meaning from previous uses of the keyword `static`.) The class declarations defining these shared variables would be something like the following:

Class declarations with static data members

```
class Aircraft {
public:
    Aircraft();
    …
private:
    static int     sMinHeight;
    int            fTime;
    PlaneData      fData;
};

class Window {
public:
    …
private:
    static Screen         sScreen;
    static Display        sDisplay;
    GC                    fGC;
    XRectangle            fRect;
    …
};
```

(As usual, it is helpful to have some naming convention. Here, static data members of classes will be given names starting with 's'.)

Defining the static variables

The class declarations specify that these variables will exist somewhere, but they don't define the variables. The definitions have to appear elsewhere. So, in the case of class `Aircraft`, the header file would contain the class declaration specifying the existence of the class data member `sMinHeight`, the definition would appear in the Aircraft.cp implementation file:

```
#include "Aircraft.h"
```

```
int Aircraft::sMinHeight = 1000; // initialize to safe 1000' value

…
int Aircraft::TooLow()
{
    return (fData.z < sMinHeight);
}
```

The definition must use the full name of the variable; this is the member name qualified by the class name, so `sMinHeight` has to be defined as `Aircraft::sMinHeight`. The `static` qualifier should not be repeated in the definition. The definition can include an initial value for the variable.

The example `TooLow()` function illustrates use of the `static` data member from inside a member function.

Quite often, such `static` variables need to be set or read by code that is not part of any of the member functions of the class. For example, the code of the `AirController` class would need to change the minimum safe height. Since the variable `sMinHeight` is private, a public access function must be provided:

```
void Aircraft::SetMinHeight(int newmin) { sMinHeight = newmin;
}
```

Static member functions

Most of the time the `AirController` worked with individual aircraft asking them to perform operations like print their details: `fAircraft[i]->PrintOn(cout)`. But when the `AirController` has to change the minimum height setting, it isn't working with a specific `Aircraft`. It is working with the `Aircraft` class as a whole. Although it is legal to have a statement like `fAircraft[i]->SetMinHeight(600)`, this isn't appropriate because the action really doesn't involve `fAircraft[i]` at all.

A member function like `SetMinHeight()` that only operates on `static` (class) data members should be declared as a `static` function:

Class declarations with static data and function members

```
class Aircraft {
public:
    Aircraft();
    …
    static void   SetMinHeight(int newmin);
private:
    static int    sMinHeight;
    int           fTime;
    PlaneData     fData;
};
```

This allows the function to be invoked by external code without involving a specific instance of class `Aircraft`, instead the call makes clear that it is "asking the class as a whole" to do something.

Calling a static member function

```
void AirController::ChangeHeight()
{
    int h;
    cout << "What is the new minimum? ";
    cin >> h;
    if((h < 300) || (h > 1500)) {
           cout << "Don't be silly" << endl; return;
           }
    Aircraft::SetMinHeight(h);
}
```

Use of statics

You will find that most of the variables that you might initial think of as being "globals" will be better defined as `static` members of one or other of the classes in your program.

One fairly common use is getting a unique identifier for each instance of a class:

```
class Thing {
public:
    Thing();
    …
private:
    static int    sIdCounter;
    int    fId;
    …
};

int Thing::sIdCounter = 0;

Thing::Thing() { fId = ++sIdCounter; … }
```

Each instance of class `Thing` has its own identifier, `fId`. The `static` (class) variable `sIdCounter` gets incremented every time a new `Thing` is created and so its value can serve as the latest `Thing`'s unique identifier.

23.2 FRIENDS

As noted in the introduction to this chapter, the main use of "friend" functions will be to help build groups (clusters) of classes that need to work closely together.

In Chapter 21, we had class `BinaryTree` that used a helper class, `TreeNode`. `BinaryTree` created `TreeNodes` and got them to do things like replace their keys. Other parts of the program weren't supposed to use `TreeNodes`. The example in Chapter 21 hid the `TreeNode` class inside the implementation file of `BinaryTree`. The header file defining class `BinaryTree` merely had the declaration `class TreeNode;` which simply allowed it to refer to `TreeNode*` pointers. This arrangement prevents other parts of a program from using `TreeNodes`. However, there are times when you can't arrange the implementation like that; code for the main class (equivalent to

BinaryTree) might have to be spread over more than one file. Then, you have to properly declare the auxiliary class (equivalent of TreeNode) in the header file. Such a declaration exposes the auxiliary class, opening up the chance that instances of the auxiliary class will get used inappropriately by other parts of the program.

This problem can be resolved using a friend relation as follows:

A very private class

```
class Auxiliary {
    friend class MainClass;
private:
    Auxiliary();
    int     SetProp1(int newval);
    void    PrintOn(ostream&) const;
    …
    int     fProp1;
    …
};
```

that has a friend

```
class MainClass {
public:
    …
};
```

All the member functions and data members of class Auxiliary are declared private, even the constructor. The C++ compiler will systematically enforce the private restriction. If it finds a variable declaration anywhere in the main code, e.g. Auxiliary a1;, it will note that this involves an implicit call to the constructor Auxiliary::Auxiliary() and, since the constructor is private, the compiler will report an access error. Which means that you can't have any instances of class Auxiliary!

However, the friend clause in the class declaration partially removes the privacy wall. Since class MainClass is specified to be a friend of Auxiliary, member functions of MainClass can invoke any member functions (or data members) of an Auxiliary object. Member functions of class MainClass can create and use instances of class Auxiliary.

There are other uses of friend relations but things like this example are the main ones. The friend relation is being used to selectively "export" functionality of a class to chosen recipients.

23.3 ITERATORS

With collection classes, like those illustrated in Chapter 21, it is often useful to be able to step through the collection processing each data member in turn. The member functions for List and DynamicArray did allow for such iterative access, but only in a relatively clumsy way:

```
DynamicArray      d1;
…
…
for(int i = 1; i < d1.Length(); i++) {
    Thing* t = (Thing*) d1.Nth(i);
    t->DoSomething();
    …
    }
```

That code works OK for `DynamicArray` where `Nth()` is basically an array indexing operation, but it is inefficient for `List` where the `Nth()` operation involves starting at the beginning and counting along the links until the desired element is found.

The `PrintOn()` function for `BinaryTree` involved a "traversal" that in effect iterated though each data item stored in the tree (starting with the highest key and working steadily to the item with the lowest key). However the `BinaryTree` class didn't provide any general mechanism for accessing the stored elements in sequence.

Mechanisms for visiting each data element in turn could have been incorporated in the classes. The omission was deliberate.

Increasingly, program designers are trying to generalize, they are trying to find mechanisms that apply to many different problems. General approaches have been proposed for working through collections.

The basic idea is to have an "Iterator" associated with the collection (each collection has a specialized form of Iterator as illustrated below). An Iterator is in itself a simple class. Its public interface would be something like the following (function names may differ and there may be slight variations in functionality):

```
class Iterator {
public:
    Iterator(…);
    void   First(void);
    void   Next(void);
    int    IsDone(void);
    void   *CurrentItem(void);
private:
    …
};
```

The idea is that you can create an iterator object associated with a list or tree collection. Later you can tell that iterator object to arrange to be looking at the "first" element in the collection, then you can loop examining the items in the collection, using `Next()` to move on to the next item, and using the `IsDone()` function to check for completion:

```
Collection c1;
…
Iterator   i1(c1);
i1.Start();
while(!i1.IsDone()) {
```

```
    Thing* t = (Thing*) i1.CurrentItem();
    t->DoSomething();
    …;
    i1.Next();
    }
```

This same code would work whether the collection were a `DynamicArray`, a `List`, or a `BinaryTree`.

An "abstract base class" for Iterators?

As explained in the final section of this chapter, it is possible to start by giving an abstract definition of an iterator as a "pure abstract class", and then define derived subclasses that represent specialized iterators for different types of collection. Here, we won't bother to define the general abstraction, and will just define and use examples of specialized classes for the different collections.

Insecure iterators

The iterators illustrated here are "insecure". If a collection gets changed while an iterator is working, things can go wrong. (There is an analogy between an iterator walking along a list and a person using stepping stones to cross a river. The iterator moves from listcell to listcell in response to `Next()` requests; it is like a person stepping onto the next stone and stopping after each step. Removal of the listcell where the iterator is standing has an effect similar to magically removing a stepping stone from under the feet of the river crosser.) There are ways of making iterators secure, but they are too complex for this introductory treatment.

23.3.1 ListIterator

An iterator for class `List` is quite simple to implement. After all, it only requires a pointer to a listcell. This pointer starts pointing to the first listcell, and in response to "Next" commands should move from listcell to listcell. The code implementing the functions for `ListIterator` is so simple that all its member functions can be defined "inline".

Consequently, adding an iterator for class `List` requires only modification of the header file:

```
#ifndef __MYLIST__
#define __MYLIST__

class ListIterator;

class List {
public:
    List();

    int           Length(void) const;
    …
    friend class ListIterator;
private:
```

Nominate friends

Special access function only used by friends

```
    struct ListCell { void *fData; ListCell *fNext; };
    ListCell *Head(void) const;

    int           fNum;
    ListCell      *fHead;
    ListCell      *fTail;
};
```

Declare the iterator class

```
class ListIterator {
public:
    ListIterator(List *l);
    void   First(void);
    void   Next(void);
    int    IsDone(void);
    void   *CurrentItem(void);
private:
    List::ListCell *fPos;
    List          *fList;
};

inline int List::Length(void) const { return fNum; }
inline List::ListCell *List::Head() const { return fHead; }
```

Implementation of ListIterator

```
inline ListIterator::ListIterator(List *l)
    { fList = l; fPos = fList->Head(); }
inline void ListIterator::First(void) { fPos = fList->Head(); }
inline void ListIterator::Next(void)
    { if(fPos != NULL) fPos = fPos->fNext; }
inline int ListIterator::IsDone(void) { return (fPos == NULL);
}
inline void *ListIterator::CurrentItem(void)
    { if(fPos == NULL) return NULL; else return fPos->fData; }
#endif
```

Friend nomination

There are several points to note in this header file. Class `List` nominates class `ListIterator` as a friend; this means that in the code of class `ListIterator`, there can be statements involving access to private data and functions of class `List`.

Access function List::Head()

Here, an extra function is defined – `List::Head()`. This function is private and therefore only useable in class `List` and its friends (this prevents clients from getting at the head pointer to the chain of listcells). Although, as a friend, a `ListIterator` can directly access the `fHead` data member, it is still preferable that it use a function style interface. You don't really want friends becoming too intimate for that makes it difficult to locate problems if something goes wrong.

Declaration of ListIterator class

The class declaration for `ListIterator` is straightforward except for the type of its `fPos` pointer. This is a pointer to a `ListCell`. But the struct `ListCell` is defined within class List. If, as here, you want to refer to this data type in code outside of that of class `List`, you must give its full type name. This is a `ListCell` as defined by class `List`. Hence, the correct type name is `List::ListCell`.

Implementation of ListIterator

The member functions for class `ListIterator` are all simple. The constructor keeps a pointer to the `List` that it is to work with, and initializes the `fPos` pointer to the first listcell in the list. Member function `First()` resets the pointer (useful if you want the iterator to run through the list more than once); `Next()` advances the pointer; `CurrentItem()` returns the data pointer from the current listcell; and `IsDone()` checks whether the `fPos` pointer has advanced off the end of the list and become `NULL`. (The code for `Next()` checks to avoid falling over at the end of a list by being told to take the "next" of a `NULL` pointer. This could only occur if the client program was in error. You might choose to "throw an exception", see Chapter 26, rather than make it a "soft error".)

The test program used to exercise class `List` and class `DynamicArray` can be extended to check the implementation of class `ListIterator`:. It needs a new branch in its `switch()` statement, one that allows the tester to request that a `ListIterator` "walk" along the `List`:

```
case 'w':
            {
            ListIterator li(&c1);
            li.First();
            cout << "Current collection " << endl;
            while(!li.IsDone()) {
                  Book p = (Book) li.CurrentItem();
                  cout << p << endl;
                  li.Next();
                  }
            }
            break;
```

The statement:

```
ListIterator li(&c1);
```

creates a `ListIterator`, called `li`, giving it the address of the `List`, `cl`, that it is to work with (the `ListIterator` constructor specifies a pointer to `List`, hence the need for an `&` address of operator).

The statement, `li.First()`, is redundant because the constructor has already performed an equivalent initialization. It is there simply because that is the normal pattern for walking through a collection:

```
li.First();
while(!li.IsDone()) {
    … li.CurrentItem();
    …
    li.Next();
    }
```

Note the need for the typecast:

```
Book p = (Book) li.CurrentItem();
```

In the example program, `Book` is a pointer type (actually just a `char*`). The `CurrentItem()` function returns a `void*`. The programmer knows that the only things that will be in the `cl` list are `Book` pointers; so the type cast is safe. It is also necessary because of course you can't really do anything with a `void*` and here the code needs to process the books in the collection.

Backwards and forwards iterators in two way lists

Class `List` is singly linked, it only has "next" pointers in its listcells. This means that it is only practical to "walk forwards" along the list from the head to the tail. If the list class uses listcells with both "next" and "previous" pointers, it is practical to walk the list in either direction. Iterators for doubly linked lists usually take an extra parameter in their constructor; this is a "flag" that indicates whether the iterator is a "forwards iterator" (start at the head and follow the next links) or a "backwards iterator" (start at the tail and follow the previous links).

23.3.2 TreeIterator

Like doubly linked lists that can have forwards or backwards iterators, binary trees can have different kinds of iterator. An "in order" iterator process the left subtree, handles the data at a treenode, then processes the right subtree; a "pre order" iterator processes the data at a tree node before examining the left and right subtrees. However, if the binary tree is a search tree, only "in order" traversal is useful. An in order style of traversal means that the iterator will return the stored items in increasing order by key.

An iterator that can "walk" a binary tree is a little more elaborate than that needed for a list. It is easy to descend the links from the root to the leaves of a tree, but there aren't any "back pointers" that you could use to find your way back from a leaf to the root. Consequently, a `TreeIterator` can't manage simply with a pointer to the current `TreeNode`, it must also maintain some record of information describing how it reached that `TreeNode`.

Stack of pointers maintain state of traversal

As illustrated in Figure 23.1, the iterator uses a kind of "stack" of pointers to `TreeNodes`. In response to a `First()` request, it chases down the left vine from the root to the left most leaf; so, in the example shown in Figure 23.1 it stacks up pointers to the `TreeNodes` associated with keys 19, 12, 6.

A `CurrentItem()` request should return the data item associated with the entry at the top of this stack.

A `Next()` request has to replace the topmost element by its successor (which might actually already be present in the stack). As illustrated in Figure 23.1, the `Next()` request applied when the iterator has entries for 19, 12, and 6, should remove the 6 and add entries for 9 and 7.

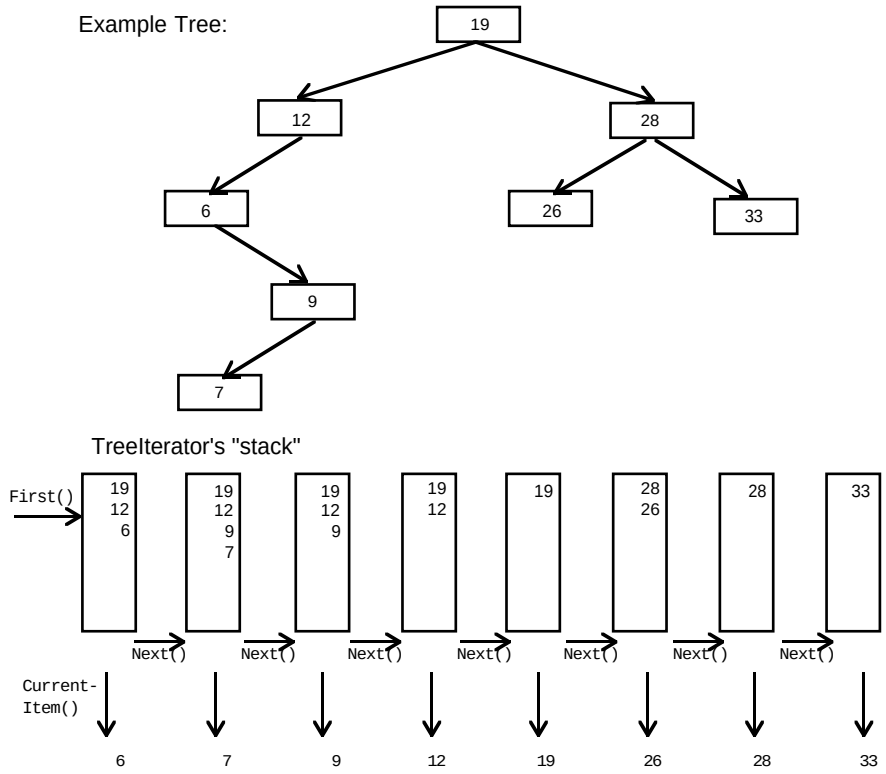

Figure 23.1 Tree and tree iterator.

A subsequent `Next()` request removes the 7, leaving 19, 12, and 9 on the stack. Further `Next()` requests remove entries until the 19 is removed, it has to be replaced with its successor so then the stack is filled up again with entries for 28 and 26.

Representing the stack

The programmer implementing class `TreeIterator` has to chose how to represent this stack. If you wanted to be really robust, you would use a `DynamicArray` of `TreeNode` pointers, this could grow to whatever size was needed. For most practical purposes a fixed size array of pointers will suffice, for instance an array with one hundred elements. The size you need is determined by the maximum depth of the tree and thus depends indirectly on the number of elements stored in the tree. If the tree were balanced, a depth of one hundred would mean that the tree had quite a large number of nodes (something like 2^{99}). Most trees are poorly balanced. For example if you inserted 100 data items into a tree in decreasing order of their keys, the left branch would be one hundred deep. Although a fixed array will do, the code needs to check for the array becoming full.

Class `BinaryTree` has to nominate class `TreeIterator` as a "friend", and again for style its best to provide a private access function rather than have this friend rummage around in the data:

```
class BinaryTree
{
public:
    BinaryTree();
    …
    friend class TreeIterator;
private:
    TreeNode      *Root(void);
    …
};

inline TreeNode *BinaryTree::Root(void)  { return fRoot; }
```

Class `TreeIterator` has the standard public interface for an iterator; its private data consist of a pointer to the `BinaryTree` it works with, an integer defining the depth of the "stack", and the array of pointers:

```
class TreeIterator {
public:
    TreeIterator(BinaryTree *tree);
    void   First(void);
    void   Next(void);
    int    IsDone(void);
    void   *CurrentItem(void);
private:
    int           fDepth;
    TreeNode      *fStack[kITMAXDEPTH];
    BinaryTree    *fTree;
};
```

The constructor simply initializes the pointer to the tree and the depth counter. This initial value corresponds to the terminated state, as tested by the `IsDone()` function. For this iterator, a call to `First()` must be made before use.

```
TreeIterator::TreeIterator(BinaryTree *tree)
{
    fTree = tree;
    fDepth = -1;
}

int TreeIterator::IsDone(void)
{
    return (fDepth < 0);
}
```

Function `First()` starts at the root and chases left links for as far as it is possible to go; each `TreeNode` visited during this process gets stacked up. This process gets things set up so that the data item with the smallest key will be the one that gets fetched first.

```
void TreeIterator::First(void)
{
    fDepth = -1;
    TreeNode *ptr = fTree->Root();
    while(ptr != NULL) {
           fDepth++;
           fStack[fDepth] = ptr;
           ptr = ptr->LeftLink();
           }
}
```

Data items are obtained from the iterator using `CurrentItem()`. This function just returns the data pointer from the `TreeNode` at the top of the stack:

```
void *TreeIterator::CurrentItem(void)
{
    if(fDepth < 0) return NULL;
    else
     return fStack[fDepth]->Data();
}
```

The `Next()` function has to "pop" the top element (i.e. remove it from the stack) and replace it by its successor. Finding the successor involves going down the right link, and then chasing left links as far as possible. Again, each `TreeNode` visited during this process gets "pushed" onto the stack. (If there is no right link, the effect of `Next()` is merely to pop an element from the stack.)

```
void TreeIterator::Next(void)
{
    if(fDepth < 0) return;

    TreeNode *ptr = fStack[fDepth];
    fDepth--;
    ptr = ptr->RightLink();
    while(ptr != NULL) {
           fDepth++;
           fStack[fDepth] = ptr;
           ptr = ptr->LeftLink();
           }
}
```

Use of the iterator should be tested. An additional command can be added to the test program shown previously:

```
case 'w':
            {
            TreeIterator ti(&gTree);
            ti.First();
            cout << "Current tree " << endl;
            while(!ti.IsDone()) {
                  DataItem *d = (DataItem*) ti.CurrentItem();
                  d->PrintOn(cout);
                  ti.Next();
                  }
            }
            break;
```

23.4 OPERATOR FUNCTIONS

Those `Add()`, `Subtract()`, and `Multiply()` functions in class `Number` (Chapter 19) seem a little unaesthetic. It would be nicer if you could write code like the following:

```
Number a("9741762756765432657365436586523454236387426 6");
Number b("65765463658764538654137245665");
Number c;
c = a + b;
```

The operations '+', '-', '/' and '*' have their familiar meanings and `c = a + b` does read better than `c = a.Add(b)`. Of course, if you are going to define '+', maybe you should define `++`, `+=`, `--`, `-=`, etc. If you do start defining operator functions you may have quite a lot of functions to write.

Operator functions are overrated. There aren't that many situations where the operators have intuitive meanings. For example you might have some "string" class that packages C-style character strings (arrays each with a '\0' terminating character as its last element) and provides operations like Concatenate (append):

```
String a("Hello");
String b(" World");

c = a.Concatenate(b);              // or maybe?  c = a + b;
```

You could define a '+' operator to work for your string class and have it do the concatenate operation. It might be obvious to you that + means "append strings", but other people won't necessarily think that way and they will find your `c = a + b` more difficult to understand than `c = a.Concatenate(b)`.

When you get to use the graphics classes defined in association with your IDE's framework class library, you will find that they often have some operator functions defined. Thus class `Point` may have an `operator+` function (this will do something

like vector addition). Or, you might have class `Rectangle` where there is an "`operator+(const Point&)`" function; this curious thing will do something like move the rectangle's topleft corner by the x, y amount specified by the `Point` argument (most people find it easier if the class has a `Rectangle::MoveTopLeftCorner()` member function).

Generally, you should not define operator functions for your classes. You can make exceptions for some. Class `Number` is an obvious candidate. You might be able to pretty up class `Bitmap` by giving it "And" and "Or" functions that are defined in terms of operators.

Apart from a few special classes where you may wish to define several operator functions, there are a couple of operators whose meanings you have to redefine in many classes.

23.4.1 Defining operator functions

As far as a compiler is concerned, the meaning of an operator like '+' is defined by information held in an internal table. This table will specify the code that has to be generated for that operator. The table will have entries like:

```
operator context            translation
long     +    long     load integer register with first data item
                       add second data item to contents of register

double   +    double   load floating point register with first data item
                       add second data item to contents of register
```

The translation may specify a sequence of instructions like those shown. But some machines don't have hardware for all arithmetic operations. There are for example RISC computers that don't have "floating point add" and "floating point multiply"; some don't even have "integer divide". The translations for these operators will specify the use of a function:

```
operator context            translation
long     /    long     push dividend and divisor onto stack
                       call ".div" function
```

In most languages, the compiler's translation tables are fixed. C++ allows you to add extra entries. So, if you have some "add" code for a class `Point` that you've defined and you want this called for `Point + Point`, you can specify this to the compiler. It takes details from your specification and appends these to its translation tables:

```
operator context            translation
point    +    point    push the two points onto the stack
                       call the function defined by the programmer
```

The specifications that must appear in your classes are somewhat unpronounceable. An addition operator would be defined as the function:

```
operator+()
```

(say that as "operator plus function"). For example, you could have:

```
class Point {
public:
    Point();
    …
    Point operator+(const Point& other) const;
    …
private:
    int    fh, fv;
};
```

with the definition:

```
Point Point::operator+(const Point& other) const
{
    Point vecSum;
    vecSum.fh = this->fh + other.fh;
    vecSum.fv = this->fv + other.fv;
    return vecSum;
}
```

This example assumes that the `+` operation shouldn't change either of the `Points` that it works on but should create a temporary `Point` result (in the return part of a function stackframe) that can be used in an assignment; this makes it like `+` for integers and doubles.

It is up to you to define the meaning of operator functions. Multiplying points by points isn't very meaningful, but multiplying points by integers is equivalent to scaling. So you *could* have the following where there is a multiply function that changes the `Point` object that executes it:

```
class Point {
public:
    Point();
    …
    Point operator+(const Point& other) const;
    Point& operator*(int scalefactor);
    …
private:
    int    fh, fv;
};
```

with a definition:

```
Point& Point::operator*(int scalefactor)
{
    // returning a reference allows expressions that have
    // scaling operations embedded inside them.
    fh *= scalefactor;
    fv *= scalefactor;
    return *this;
}
```

with these definitions you can puzzle anyone who has to read and maintain your code by having constructs like:

```
Point a(6,4);

…;
a*3;

Point b(7, 2);
Point c;
…
c = b + a*4;
```

Sensible maintenance programmers will eventually get round to changing your code to:

```
class Point {
public:
    Point();
    …
    Point operator+(const Point& other) const;
    void   ScaleBy(int scalefactor);
    …
};

void Point::ScaleBy(int scalefactor)
{
    fh *= scalefactor;
    fv *= scalefactor;
}
```

resulting in more intelligible programs:

```
Point a(6,4);

…;
a.ScaleBy(3);
```

```
Point b(7, 2);
Point c;
…
a.ScaleBy(4);
c = b + a;
```

Avoid the use of operator functions except where their meanings are universally agreed. If their meanings are obvious, operator function can result in cosmetic improvements to the code; for example, you can pretty up class `Number` as follows:

```
class Number {
public:
    // Member functions declared as before
    …
    Number operator+(const Number& other) const;
    …
    Number operator/(const Number& other) const;
private:
    // as before
    …
};

inline Number Number::operator+(const Number& other) const
{
    return this->Add(other);
}
```

Usually, the meanings of operator functions are not obvious

23.4.2 Operator functions and the iostream library

You will frequently want to extend the meanings of the << and >> operators. A C++ compiler's built in definition for these operators is quite limited:

operator context			**translation**
long	**<<**	long	load integer register with *first data item* shift left by the specified number of places
long	**>>**	long	load integer register with *first data item* shift right by the specified number of places

But if you #include the iostream header files, you add all the "takes from" and "gives to" operators:

operator context			**translation**
ostream	**<<**	long	push the ostream id and the long onto the stack call the function "ostream::operator<<(long)"

```
istream >>   long     push the istream id and the address of the long
                            onto the stack
                      call the function "istream::operator>>(long&)"
```

These entries are added to the table as the compiler reads the iostream header file with its declarations like:

```
class ostream {
public:
    …
    ostream& operator<<(long);
    ostream& operator<<(char*);
    …
};
```

Such functions declared in the iostream.h header file are member functions of class `istream` or class `ostream`. An `ostream` object "knows" how to print out a long integer, a character, a double, a character string and so forth.

How could you make an `ostream` object know how to print a `Point` or some other programmer defined class?

Typically, you will already have defined a `PrintOn()` member function in your `Point` class.

```
class Point {
public:
    …
    void PrintOn(ostream& out);
private:
    int    fh, fv;
};

void Point::PrintOn(ostream& out)
{
    out << "(" << fh << ", " << fv << ") ";
}
```

and all you really want to do is make it possible to write something like:

```
Point p1, p2;
…
cout << "Start point " << p1 << ", end point " << p2 << endl;
```

rather than:

```
cout << "Start point ";
p1.PrintOn(cout);
cout << ", end point ";
```

```
p2.PrintOn(cout);
cout << endl;
```

You want someway of telling the compiler that if it sees the `<<` operator involving an `ostream` and a `Point` then it is to use code similar to that of the `Point::PrintOn()` function (or maybe just use a call to the existing `PrintOn()` function).

You *could* change the classes defined in the iostream library. You could add extra member functions:

```
class ostream {
// everything as now plus
    ostream& operator<<(const Point& p);
    …
};
```

and provide your definition of `ostream& ostream::operator<<(const Point&)`.

It should be obvious that this is not desirable. The iostream library has been carefully developed and debugged. You wouldn't want hundreds of copies each with minor extensions hacked in.

Fortunately, such changes aren't necessary. There is another way of achieving the desired effect.

A global operator<<(ostream& , Point&) function

You can define global operator functions. These functions aren't members of classes. They are simply devices for telling the compiler how it is to translate cases where it finds an operator involving arguments of specified types.

In this case, you need to define a new meaning for the `<<` operator when it must combine an `ostream` and a `Point`. So you define:

```
?? operator<<(ostream& os, const Point& p)
{
    p.PrintOn(os);
    return ??
}
```

(the appropriate return type will be explained shortly). The compiler invents a name for the function (it will be something complex like `__leftshift_Tostreamref_cTPointref`) and adds the new meaning for `<<` to its table:

operator context		**translation**
ostream **<<**	point	push the ostream id and the point's address onto the stack call the function __leftshift_Tostreamref_cTPointref

This definition then allows constructs like: `Point p; …; cout << p;`.

Of course, the ideal is for the stream output operations to be concatenated as in:

```
cout << "Start point " << p1 << ", end point " << p2 << endl;
```

This requirement defines the return type of the function. It must return a reference to the ostream:

```
ostream& operator<<(ostream& os, const Point& p)
{
    p.PrintOn(os);
    return os;
}
```

Having a reference to the stream returned as a result permits the concatenation. Figure 23.2 illustrates the way that the scheme works.

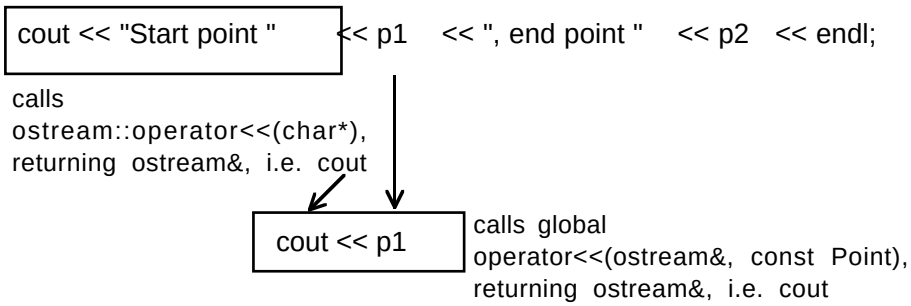

Figure 23.2 Illustration of groupings involved in concatenated use of ostream& operator<<() functions.

You might also want:

```
ostream& operator<<(ostream& os, Point *p_pt)
{
    p_pt->PrintOn(os);
    return os;
}
```

23.5 RESOURCE MANAGER CLASSES AND DESTRUCTORS

This section explains some of the problems associated with "resource manager" classes.

Resource manager classes are those whose instances own other data structures. Usually, these will be other data structures separately allocated in the heap. We've already seen examples like class `DynamicArray` whose instances each own a separately allocated array structure. However, sometimes the separately allocated data structures

may be in operating system's area; examples here are resources like open files, or "ports and sockets" as used for communications between programs running on different computers.

The problems for resource managers are:

- disposal of managed resources that are no longer required;
- unintended sharing of resources.

The first subsection, 23.5.1, provides some examples illustrating these problems. The following two sections present solutions.

23.5.1 Resource management

Instances of classes can acquire resources when they are created, or as a result of subsequent actions. For example, an object might require a variable length character string for a name:

```
class DataItem {
public:
    DataItem(const char* dname);
    …
private:
    char   *fName;
    …
};

DataItem::DataItem(const char* dname)
{
    fName = new char[strlen(dname) + 1];
    strcpy(fName, dname);
    …
}
```

Another object might need to use a file:

```
class SessionLogger {
public:
    SessionLogger();
    …
    int    OpenLogFile(const char* logname);
    …
private:
    …
    ofstream      fLfile;
    …
};
```

```
int SessionLogger::OpenLogFile(const char* logname)
{
    fLfile.open(logname, ios::out);
    return fLfile.good();
}
```

Instances of the `DataItem` and `SessionLogger` classes will be created and destroyed in various ways:

```
void DemoFunction()
{
    while(AnotherSession()) {
            char   name[100];
            cout << "Session name: "; cin >> name;
            SessionLogger s1;
            if(0 == s1.OpenLogFile(name)) {
                   cout << "Can't continue, no file.";
                   break;
                   }
            for(;;) {
                   char   dbuff[100];
                   …
                   DataItem      *dptr = new DataItem(dbuff);
                   …
                   delete dptr;
                   }
            }
}
```

In the example code, a `SessionLogger` object is, in effect, created in the stack and subsequently destroyed for each iteration of the `while` loop. In the enclosed `for` loop, `DataItem` objects are created in the heap, and later explicitly deleted.

Figure 23.3 illustrates the representation of a `DataItem` (and its associated name) in the heap, and the effect of the statement `delete dptr`. As shown, the space occupied by the primary `DataItem` structure itself is released; but the space occupied by its name string remains "in use". Class `DataItem` has a "memory leak".

Figure 23.4 illustrates another problem with class `DataItem` , this problem is sharing due to assignment. The problem would show up in code like the following (assume for this example that class `DataItem` has member functions that change the case of all letters in the associated name string):

```
DataItem d1("This One");
DataItem d2("another one");
…
d2 = d1;
```

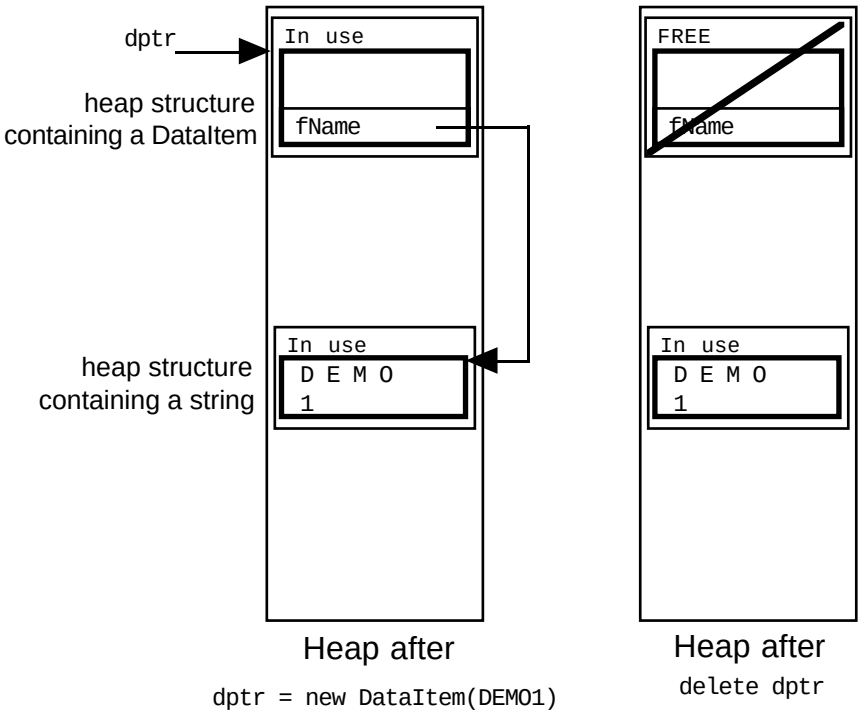

Figure 23.3 Resource manager class with memory leak.

```
…
d1.MakeLowerCase();
d2.MakeUpperCase();
d1.PrintOn(cout);
…
```

The assignment `d2 = d1` will work by default. The contents of record structure `d1` are copied field by field into `d2`, so naturally `d2`'s `fName` pointer is changed to point to the same string as that referenced by `d1.fName`. (There is also another memory leak; the string that used to be owned by `d2` has now been abandoned.)

Since `d2` and `d1` both share the same string, any operations that they perform on that string will interact. Although object `d1` has made its string lower case, `d2` changes it to upper case so that when printed by `d1` it appears as upper case.

Class `SessionLogger` has very similar problems. The resource that a `SessionLogger` object owns is some operating system structure describing a file. Such structures, lets just call them "file descriptors," get released when files are closed. If a `SessionLogger` object is destroyed before it closes its associated file, the file descriptor structures remain. When a program finishes, all files are closed and the associated file descriptor structures are released.

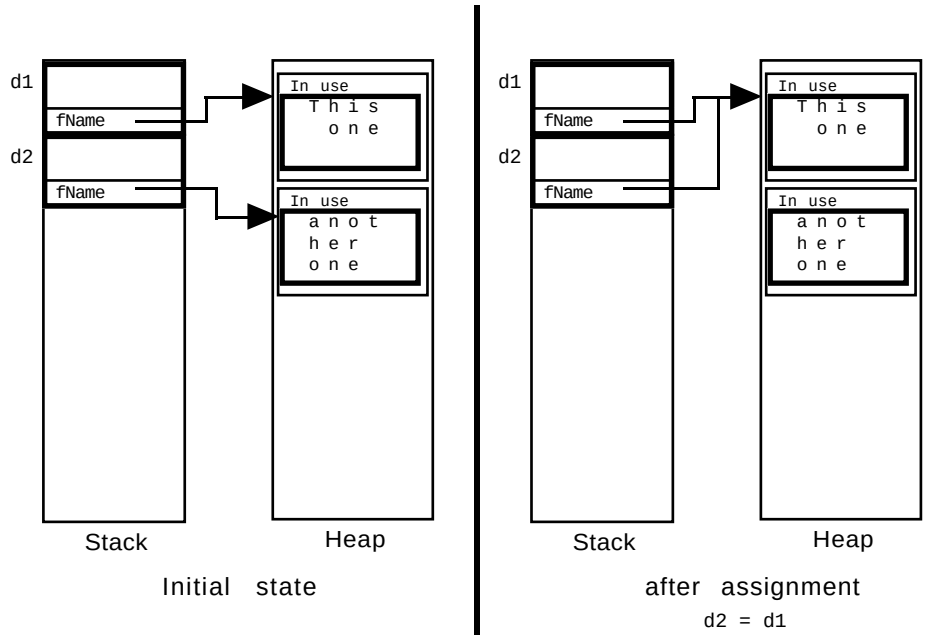

Figure 23.4 Assignment leading to sharing of resources.

However, an operating system normally limits the number of file descriptors that a program can own. If `SessionLogger` objects don't close their files, then eventually the program will run out of file descriptors (its a bit like running out of heap space, but you can make it happen a lot more easily).

Structure sharing will also occur if a program's code has assignment statements involving `SessionLoggers`:

```
SessionLogger s1, s2;
…
s1.OpenLogFile("testing");
…
s2 = s1;
```

Both `SessionLogger` objects use the same file. So if one does something like cause a seek operation (explicitly repositioning the point where the next write operation should occur), this will affect the other `SessionLogger`.

23.5.2 Destructor functions

Some of the problems just explained can be solved by arranging that objects get the chance to "tidy up" just before they themselves get destroyed. You *could* attempt to achieve this by hand coding. You would define a "TidyUp" function in each class:

```
void DataItem::TidyUp() { delete [] fName; }

void SessionLogger::TidyUp() { fLfile.close(); }
```

You would have to include explicit calls to these `TidyUp()` functions at all appropriate points in your code:

```
while(AnotherSession()) {
    …
    SessionLogger s1;
    …
    for(;;) {
            …
            DataItem      *dptr = new DataItem(dbuff);
            …
            dptr->TidyUp();
            delete dptr;
            }
    s1.TidyUp();
    }
```

That is the problem with "hand coding". It is very easy to miss some point where an automatic goes out of scope and so forget to include a tidy up routine. Insertion of these calls is also tiresome, repetitious "mechanical" work.

Tiresome, repetitious "mechanical" work is best done by computer program. The compiler program can take on the job of putting in calls to "TidyUp" functions. Of course, if the compiler is to do the work, things like names of functions have to be standardized.

For each class you can define a "destructor" routine that does this kind of tidying up. In order to standardize for the compiler, the name of the destructor routine is based on the class name. For class `X`, you had constructor functions, e.g. `X()`, that create instances, and you can have a destructor function `~X()` that does a tidy up before an object is destroyed. (The character ~, "tilde", is the symbol used for NOT operations on bit maps and so forth; a destructor is the NOT, or negation, of a constructor.)

Rather than those "TidyUp" functions, class `DataItem` and class `SessionLogger` would both define destructors:

```
class DataItem {
public:
    DataItem(const char *name);
```

```
    ~DataItem();
    …
};

DataItem::~DataItem() { delete [] fName; }

class SessionLogger {
public:
    SessionLogger();
    ~SessionLogger() { this->fLfile.close(); }
    …
};
```

Just as the compiler put in the implicit calls to constructor functions, so it puts in the calls to destructors.

You can have a class with several constructors because there may be different kinds of data that can be used to initialize a class. There can only be one destructor; it takes no arguments. Like constructors, a destructor has no return type.

Destructors can exacerbate problems related to structure sharing. As we now have a destructor for class `DataItem`, an individual `DataItem` object will dutifully delete its name when it gets destroyed. If assignment has lead to structure sharing, there will be a second `DataItem` around whose name has suddenly ceased to exist.

You don't have to define destructors for all your classes. Destructors are needed for classes that are themselves resource managers, or classes that are used as "base classes" in some class hierarchy (see section 23.6).

Several of the collection classes in Chapter 21 were resource managers and they should have had destructors.

Class `DynamicArray` would be easy, it owns only a single separately allocated array, so all that its destructor need do is get rid of this:

```
class DynamicArray {
public:
    DynamicArray(int size = 10, int inc = 5);
    ~DynamicArray();

    …

private:
    …
    void   **fItems;
};

DynamicArray::~DynamicArray() { delete [] fItems; }
```

Note that the destructor does not delete the data items stored in the array. This is a design decision for all these collection classes. The collection does not own the stored items, it merely looks after them for a while. There could be other pointers to stored

items elsewhere in the program. You can have collection classes that do own the items that are stored or that make copies of the original data and store these copies. In such cases, the destructor for the collection class should run through the collection deleting each individual stored item.

Destructors for class `List` and class `BinaryTree` are a bit more complex because instances of these classes "own" many listcells and treenodes respectively. All these auxiliary structures have to be deleted (though, as already explained, the actual stored data items are not to be deleted). The destructor for these collection class will have to run through the entire linked network getting rid of the individual listcells or treenodes.

A destructor for class `List` is as follows:

```
List::~List()
{
    ListCell *ptr;
    ListCell *temp;
    ptr = fHead;
    while(ptr != NULL) {
           temp = ptr;
           ptr = ptr->fNext;
           delete temp;
           }
}
```

The destructor for class `BinaryTree` is most easily implemented using a private auxiliary recursive function:

```
BinaryTree::~BinaryTree()
{
    Destroy(fRoot);
}

void BinaryTree::Destroy(TreeNode* t)
{
    if(t == NULL)
           return;
    Destroy(t->LeftLink());
    Destroy(t->RightLink());
    delete t;
}
```

The recursive `Destroy()` function chases down branches of the tree structure. At each `TreeNode` reached, `Destroy()` arranges to get rid of all the `TreeNodes` in the left subtree, then all the `TreeNodes` in the right subtree, finally disposing of the current `TreeNode`. (This is an example of a "post order" traversal; it processes the current node of the tree after, "post", processing both subtrees.)

23.5.3 The assignment operator and copy constructors

There are two places where structures or class instances are, by default, copied using a byte by byte copy. These are assignments:

```
DataItem d1("x"), d2("y");
…
d2 = d1;
…
```

and in calls to functions where objects are passed by value:

```
void foo(DataItem dd) { … ; … ;  … }

void test()
{
    DataItem anItem("Hello world");
    …
    foo(anItem);
    …
}
```

This second case is an example of using a "copy constructor". Copy constructors are used to build a new class instance, just like an existing class instance. They do turn up in other places, but the most frequent place is in situations like the call to the function requiring a value argument.

As illustrated in section 23.5.1, the trouble with the default "copy the bytes" implementations for the assignment operator and for a copy constructor is that they usually lead to undesired structure sharing.

If you want to avoid structure sharing, you have to provide the compiler with specifications for alternative ways of handling assignment and copy construction. Thus, for `DataItem`, we would need a copy constructor that made a copy of the character string `fName`:

A copy constructor that duplicates an "owned resource"

```
DataItem::DataItem(const DataItem& other)
{
    fName = new char[strlen(other.fName) + 1];
    strcpy(fName, other.fName);
    …
}
```

Assignment operator

Though similar, assignments are a little more complex. The basic form of an `operator=` function for the example class `DataItem` would be:

```
?? DataItem::operator=(const DataItem& other)
{
    …
```

```
    delete [] fName;
    fName = new char[strlen(other.fName) + 1];
    strcpy(fName, other.fName);
    …
}
```

Plugging a memory leak

The statement:

```
delete [] fName;
```

gets rid of the existing character array owned by the `DataItem`; this plugs the memory leak that would otherwise occur. The next two statements duplicate the content of the other `DataItem`'s `fName` character array.

If you want to allow assignments at all, then for consistency with the rest of C++ you had better allow concatenated assignments:

```
DataItem d1("XXX");
DataItem d2("YYY");
DataItem d3("ZZZ";
…
d3 = d2 = d1;
```

To achieve this, you have to have the `DataItem::operator=()` function to return a reference to the `DataItem` itself:

```
DataItem& DataItem::operator=(const DataItem& other)
{
    …
    delete [] fName;
    fName = new char[strlen(other.fName) + 1];
    strcpy(fName, other.fName);
    …
    return *this;
}
```

There is a small problem. Essentially, the code says "get rid of the owned array, duplicate the other's owned array". Suppose somehow you tried to assign the value of a `DataItem` to itself; the array that has then to be duplicated is the one just deleted. Such code will usually work, but only because the deleted array remains as a "ghost" in the heap. Sooner or later the code would crash; the memory manager will have rearranged memory in some way in response to the delete operation.

You might guess that "self assignments" are rare. Certainly, those like:

```
DataItem d1("xyz");
…
d1 = d1;
```

are rare (and good compilers will eliminate statements like `d1 = d1`). However, self assignments do occur when you are working with data referenced by pointers. For example, you might have:

```
DataItem *d_ptr1;
DataItem *d_ptr2;
…
// Copy DataItem referenced by d_ptr1 into the DataItem
// referenced by pointer d_ptr2
*dptr2 = *dptr1;
```

It is of course possible that `dptr1` and `dptr2` are pointing to the same DataItem.

You have to take precautions to avoid problems with self assignments. The following arrangement (usually) works:

```
DataItem& DataItem::operator=(const DataItem& other)
{
    if(this != &other) {
           delete [] fName;
           fName = new char[strlen(other.fName) + 1];
           strcpy(fName, other.fName);
           }
    return *this;
}
```

It checks the addresses of the two `DataItems`. One address is held in the (implicit) pointer argument `this`, the second address is obtained by applying the `&` address of operator to `other`. If the addresses are equal it is a self assignment so don't do anything.

Preventing copying

Of course, sometimes it is just meaningless to allow assignment and copy constructors. You really wouldn't want two `SessionLoggers` working with the same file (and they can't really have two files because their files have to have the same name). In situations like this, what you really want to do is to prevent assignments and other copying. You can achieve this by declaring a `private` copy constructor and a `private operator=` function;

```
class SessionLogger {
public:
    SessionLogger();
    ~SessionLogger();
    …
private:
    // No assignment, no copying!
    void operator=(const SessionLogger& other);
    SessionLogger(const SessionLogger& other);
    …
};
```

You shouldn't provide an implementation for these functions. Declaring these functions as `private` means that such functions can't occur in client code. Code like `SessionLogger s1, s2; …; s2 = s1;` will result in an error message like "Cannot access `SessionLogger::_assign()` here". Obviously, such operations won't occur in the member functions of the class itself because the author of the class knows that assignment and copying are illegal. The return type of the `operator=` function does not matter in this context, so it is simplest to declare it as `void`.

Assignment and copy construction should be disabled for collection classes like those from Chapter 24, e.g.:

```
class BinaryTree {
public:
    …
private:
    void operator=(const BinaryTree& other);
    BinaryTree(const BinaryTree& other);
    …
};
```

23.6 INHERITANCE

Most of the programs that you will write in future will be "object based". You will analyze a problem, identify "objects" that will be present at run-time in your program, and determine the "classes" to which these objects belong. Then you will design the various independent classes needed, implement them, and write a program that creates instances of these classes and allows them to interact.

Independent classes? That isn't always the case.

In some circumstances, in the analysis phase or in the early stages of the design phase you will identify similarities among the prototype classes that you have proposed for your program. Often, exploitation of such similarities leads to an improved design, and sometimes can lead to significant savings in implementation effort.

23.6.1 Discovering similarities among prototype classes

Example application

Suppose that you and some colleagues had to write a "Macintosh/Windows" program for manipulating electrical circuits, the simple kinds of circuit that can be made with those "Physics is Fun" sets that ambitious parents buy to disappoint their kids at Xmas. Those kits have wires, switches, batteries, lamp-bulbs and resistors, and sometimes more. A program to simulate such circuits would need an editing component that allowed a circuit to be laid out graphically, and some other part that did all the "Ohm's Law" and "Kirchoff's Law" calculations to calculate currents and "light up" the simulated bulbs.

You have used "Draw" programs so you know the kind of interface that such a program would have. There would be a "palette of tools" that a user could use to add components. The components would include text (paragraphs describing the circuit), and circuit elements like the batteries and light bulbs. The editor part would allow the user to select a component, move it onto the main work area and then, by doubly clicking the mouse button, open a dialog window that would allow editing of text and setting parameters such as a resistance in ohms. Obviously, the program would have to let the user save a partially designed circuit to a file from where it could be restored later.

What objects might the program contain?

The objects are all pretty obvious (at least they are obvious once you've been playing this game long enough). The following are among the more important:

Objects needed

- A "document" object that would own all the data, keep track of the components added and organize transfers to and from disk.
- Various collections, either "lists" or "dynamic arrays" used to store items. Lets call them "lists" (although, for efficiency reasons, a real implementation would probably use dynamic arrays). These lists would be owned by the "document". There might be a list of "text paragraphs" (text describing the circuit), a "list of wires", a "list of resistors" and so forth.
- A "palette object". This would respond to mouse-button clicks by giving the document another battery, wire, resistor or whatever to add to the appropriate list.
- A "window" or "view" object used when displaying the circuit.
- Some "dialog" objects" used for input of parameters.
- Lots of "wire" objects.
- Several "resistor objects".
- A few "switch" objects".
- A few "lamp bulb" objects".

and for a circuit that actually does something

- At least one battery object.

For each, you would need to characterize the class and work out a list of data owned and functions performed.

During a preliminary design process your group would be right to come up with classes Battery, Document, Palette, Resistor, Switch. Each group member could work on refining one or two classes leading to an initial set of descriptions like the following:

Preliminary design ideas for classes

- class TextParagraph
  Owns:
  a block of text and a rectangle defining position in main view (window).
  Does:
  GetText() – uses a standard text editing dialog to get text changed;
  FollowMouse() – responds to middle mouse button by following mouse to reposition within view;
  DisplayText() - draws itself in view;

Rect() – returns bounding rectangle;
…
Save() and Restore() - transfers text and position details to/.from file.

- class Battery
  Owns:
  Position in view, resistance (internal resistance), electromotive force, possibly a text string for some label/name, unique identifier, identifiers of connecting wires…
  Does:
  GetVoltStuff() – uses a dialog to get voltage, internal resistance etc.
  TrackMouse() – respond to middle mouse button by following mouse to reposition within view;
  DrawBat() - draws itself in view;
  AddWire() – add a connecting wire;
  Area() – returns rectangle occupied by battery in display view;
  …
  Put() and Get() – transfers parameters to/from file.

- class Resistor
  Owns:
  Position in view, resistance, possibly a text string for some label/name, unique identifier, identifiers of connecting wires…
  Does:
  GetResistance() – uses a dialog to get resistance, label etc.
  Move() – respond to middle mouse button by following mouse to reposition within view;
  Display() - draws itself in view;
  Place() – returns area when resistor gets drawn;
  …
  ReadFrom() and WriteTo() – transfers parameters to/from file.

You should be able to sketch out pseudo code for some of the main operations. For example, the document's function to save data to a file might be something like the following:

Prototype code using instances of classes

```
Document::DoSave
    write paragraphList.Length()
    iterator i1(paragraphList)
    for i1.First(), !i1.IsDone() do
           paragraph_ptr = i1.CurrentItem();
           paragraph_ptr->Save()
           i1.Next();

    write BatteriesList.Length()
    iterator i2(BatteriesList)
    for i2.First, !i2.IsDone() do
           battery_ptr = i2.CurrentItem()
```

```
            battery_ptr->Put()

    …
```

The function to display all the data of the document would be rather similar:

```
Document::Draw
    iterator i1(paragraphList)
    for i1.First(), !i1.IsDone() do
            paragraph_ptr = i1.CurrentItem();
            paragraph_ptr->DisplayText()
            i1.Next();

    iterator i2(BatteriesList)
    for i2.First, !i2.IsDone() do
            battery_ptr = i2.CurrentItem()
            battery_ptr->DrawBat()

    …
```

Another function of "Document" would sort out which data element was being picked when the user wanted to move something using the mouse pointer:

```
Document::LetUserMoveSomething(Point mousePoint)
    iterator i1(paragraphList)
    Paragraph *pp = NULL;
    for i1.First(), !i1.IsDone() do
            paragraph_ptr = i1.CurrentItem();
            Rectangle r = paragraph_ptr->Rect()
            if(r.Contains(mousePoint) pp = paragraph_ptr;
            i1.Next();
    if(pp != NULL)
            pp->FollowMouse()
            return

    iterator i2(BatteriesList)
    battery *pb
    for i2.First, !i2.IsDone() do
            battery_ptr = i2.CurrentItem()
            Rectangle r = battery_ptr ->Area()
            if(r.Contains(mousePoint) pb = battery_ptr ;
            i2.Next();

    if(pb != NULL)
            pb->TrackMouse()
            return

…
```

Design problems?

By now you should have the feeling that there is something amiss. The design with its "batteries", "wires", "text paragraphs" seems sensible. But the code is coming out curiously clumsy and unattractive in its inconsistencies.

Batteries, switches, wires, and text paragraphs may be wildly different kinds of things, but from the perspective of "document" they actually have some similarities. They are all "things" that perform similar tasks. A document can ask a "thing" to:

- Save yourself to disk;
- Display your editing dialog;
- Draw yourself;
- Track the mouse as it moves and reposition yourself;

…

Similarities among classes

Some "things" are more similar than others. Batteries, switches, and resistors will all have specific roles to play in the circuit simulation, and there will be many similarities in their roles. Wires are also considered in the circuit simulation, but their role is quite different, they just connect active components. Text paragraphs don't get involved in the circuit simulation part. So all of them are "storable, drawable, editable" things, some are "circuit things", and some are "circuit things that have resistances".

A class hierarchy

You can represent such relationships among classes graphically, as illustrated in Figure 23.5. As shown there, there is a kind of hierarchy.

An pure "abstract" class

Class Thing captures just the concept of some kind of data element that can draw itself, save itself to file and so forth. There are no data elements defined for Thing, it is purely conceptual, purely abstract.

Concrete class TextParagraph

A TextParagraph is a particular kind of Thing. A TextParagraph does own data, it owns its text, its position and so forth. You can also define actual code specifying exactly how a TextParagraph might carry out specific tasks like saving itself to file. Whereas class Thing is purely conceptual, a TextParagraph is something pretty real, pretty "concrete". You can "see" a TextParagraph as an actual data structure in a running program.

Partially abstract class CircuitThing

In contrast, a CircuitThing is somewhat abstract. You can define some properties of a CircuitThing. All circuit elements seem to need unique identifiers, they need coordinate data defining their position, and they need a character string for a name or a label. You can even define some of the code associated with CircuitThings – for instance, you could define functions that access coordinate data.

Concrete class Wire

Wires are special kinds of CircuitThings. It is easy to define them completely. They have a few more data fields (e.g. identifiers of the components that they join, or maybe coordinates for their endpoints). It is also easy to define completely how they perform all the functions like saving their data to file or drawing themselves.

Partially abstract class Component

Components are a different specialization of CircuitThing. Components are CircuitThings that will have to be analyzed by the circuit simulation component of the program. So they will have data attributes like "resistance", and they may have many additional forms of behaviour as required in the simulation.

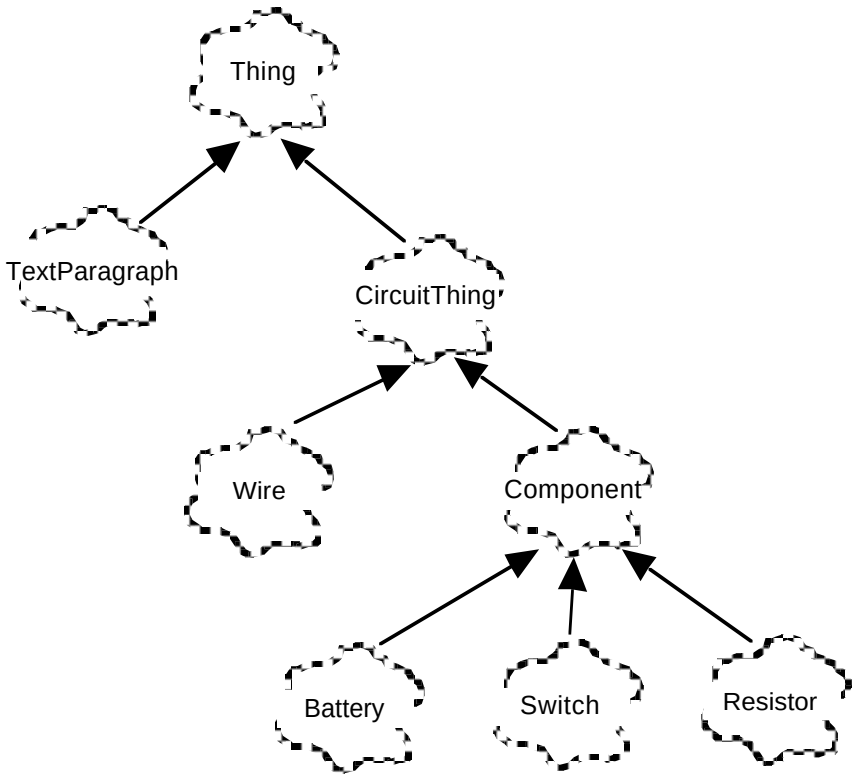

Figure 23.5 Similarities among classes.

Concrete classes Battery, Switch, …

Naturally, Battery, Switch, and Resistor define different specializations of this idea of Component. Each will have its unique additional data attributes. Each can define a real implementation for functions like Draw().

The benefits of a class hierarchy

OK, such a hierarchy provides a nice conceptual structure when talking about a program but how does it really help?

Consistency

One thing that you immediately gain is consistency. In the original design sketch, text paragraphs, batteries and so forth all had some way of defining that these data elements could display themselves, save themselves to file and so forth. But each class was slightly different; thus we had `TextParagraph::Save()`, `Battery::Put()` and `Resistor:: WriteTo()`. The hierarchy allows us to capture the concept of "storability" by specifying in class `Thing` the ability `WriteTo()`. While each

specialization performs `WriteTo()` in a unique way, they can at least be consistent in their names for this common behaviour. But consistency of naming is just a beginning.

Design simplifications

If you exploit such similarities, you can greatly simplify the design of the overall application as can be seen by re-examining some of the tasks that a Document must perform.

While you might want separate lists of the various specialized `Components` (as this might be necessary for the circuit simulation code), you could change `Document` so that it stores data using a single `thingList` instead of separate `paragraphList`, `BatteriesList` and so forth . This would allow simplification of functions like `DoSave()`:

Functions exploiting similarities

```
Document::DoSave(…)
    write thingList.Length()
    iterator i1(thingList)
    for i1.First(), !i1.IsDone() do
           thing_ptr = i1.CurrentItem();
           thing_ptr->WriteTo()
           i1.Next();

Document::Draw
    iterator i1(thingList)
    for i1.First(), !i1.IsDone() do
           thing_ptr = i1.CurrentItem();
           thing_ptr->Draw()
           i1.Next();

Document::LetUserMoveSomething(Point mousePoint)
    iterator i1(thingList)
    Thing *pt = NULL;
    for i1.First(), !i1.IsDone() do
           thing_ptr = i1.CurrentItem();
           Rectangle r = thing_ptr ->Area()
           if(r.Contains(mousePoint) pt = thing_ptr ;
           i1.Next();
    if(pt != NULL)
           pt->TrackMouse()
           return
```

The code is no longer obscured by all the different special cases. The revised code is shorter and much more intelligible.

Extendability

Note also how the revised `Document` no longer needs to know about the different kinds of circuit component. This would prove useful later if you decided to have another component (e.g. class Voltmeter); you wouldn't need to change the code of `Document` in order to accommodate this extension.

Code sharing

The most significant benefit is the resulting simplification of design, and simultaneous acquisition of extendability. But you may gain more. Sometimes, you can define the code for a particular behaviour at the level of a partially abstract class. Thus, you should be able to define the access function for getting a `CircuitThing`'s

identifier at the level of class `CircuitThing` while class `Component` can define the code for accessing a `Component`'s electrical resistance. Defining these functions at the level of the partially abstract classes saves you from writing very similar functions for each of the concrete classes like `Battery`, `Resistor`, etc.

23.6.2 DEFINING CLASS HIERARCHIES IN C++

C++ allows you to define such hierarchical relations amongst classes. So, there is a way of specifying "class `Thing` represents the abstract concept of a storable, drawable, moveable data element", "class `TextParagraph` is a kind of `Thing` that looks after text and …".

Base class

You start by defining the "base class", in this case that is class `Thing` which is the base class for the entire hierarchy:

```
class Thing {
public:
    virtual ~Thing() { }
    /* Disk I/O */
    virtual void  ReadFrom(istream& i s) = 0;
    virtual void  WriteTo(ostream& os) const = 0;
    /* Graphics */
    virtual void  Draw() const = 0;
    /* mouse interactions */
    virtual void  DoDialog() = 0;      // For double click
    virtual void  TrackMouse() = 0;    // Mouse select and drag
    virtual Rect  Area() const = 0;
    …
};
```

Class `Thing` represents just an idea of a storable, drawable data element and so naturally i t is simply a list of function names.

The situation is a little odd. We know that all `Things` can draw themselves, but we can't say how. The ability to draw is common, but the mechanism depends very much on the specialized nature of the `Thing` that is asked to draw itself. In class `Thing`, we have to be able to say "all `Things` respond to a `Draw()` request, specialized `Thing` subclasses define how they do this".

virtual keyword and =0 definition

This is what the keyword `virtual` and the odd = 0 notation are for.

Roughly, the keyword `virtual` identifies a function that a class wants to define in such a way that subclasses may later extend or otherwise modify the definition. The `=0` part means that we aren't prepared to offer even a default implementation. (Such undefined virtual functions are called "pure virtual functions".)

In the case of class `Thing`, we can't provide default definitions for any of the functions like `Draw()`, `WriteTo()` and so forth. The implementations of these functions vary too much between different subclasses. This represents an extreme case;

often you can provide a default implementation for a `virtual` function. This default definition describes what "usually" should be done. Subclasses that need to something different can replace, or "override", the default definition.

virtual destructor

The destructor, `~Thing()`, does have a definition: `virtual ~Thing() { }`. The definition is an empty function; basically, it says that by default there is no tidying up to be done when a `Thing` is deleted. The destructor is `virtual`. Subclasses of class `Thing` may be resource managers (e.g. a subclass might allocate space for an object label as a separate character array in the heap). Such specialized `Things` will need destructors that do some cleaning up.

Thing* variables

A C++ compiler prevents you from having variables of type `Thing`:

```
Thing      aThing;       // illegal, Thing is an abstraction
```

This is of course appropriate. You can't have `Things`. You can only have instances of specialized subclasses. (This is standard whenever you have a classification hierarchy with abstract classes. After all, you never see "mammals" walking around, instead you encounter dogs, cats, humans, and horses – i.e. instances of specialized subclasses of class mammal). However, you can have variables that are `Thing*` pointers, and you can define functions that take `Thing&` reference arguments:

```
Thing *first_thing;
```

The pointer `first_thing` can hold the address of (i.e. point to) an instance of class `TextParagraph`, or it might point to a `Wire` object, or point to a `Battery` object.

Derived classes

Once you have declared class `Thing`, you can declare classes that are "based on" or "derived from" this class:

Public derivation tag

```
class TextParagraph : public Thing {
    TextParagraph(Point topleft);
    virtual ~TextParagraph();
    /* Disk I/O */
    virtual void  ReadFrom(istream& is);
    virtual void  WriteTo(ostream& os) const;
    /* Graphics */
    virtual void  Draw() const;
    /* mouse interactions */
    virtual void  DoDialog();   // For double click
    virtual void  TrackMouse(); // Mouse select and drag
    virtual Rect  Area() const;
    // Member functions that are unique to TextParagraphs
    void   EditText();
    …
private:
    // Data needed by a TextParagraph
    Point  fTopLeft;
    char   *fText;
    …
```

```
};

class CircuitThing : public Thing {
    CircuitThing(int ident, Point where);
    virtual ~CircuitThing();
    …
    /* Disk I/O */
    virtual void  ReadFrom(istream& is);
    virtual void  WriteTo(ostream& os) const;
    …
    // Additional member functions that define behaviours
    // characteristic of all kinds of CircuitThing
    int           GetId() const { return this->fId }
    virtual Rect  Area() const {
           return Rect(
                  this->flocation.x - 8, this->flocation.y - 8,
                  this->flocation.x + 8, this->flocation.y + 8);
                  }
    virtual double Current() const = 0;
    …
protected:
    // Data needed by a CircuitThing
    int    fId;
    Point  flocation;
    char   *fLabel;
    …

};
```

Protected access specifier

In later studies you will learn that there are a variety of different ways that "derivation" can be used to build up class hierarchies. Initially, only one form is important. The important form is "public derivation". Both `TextParagraph` and `CircuitThing` are "*publicly derived*" from class `Thing`:

Different forms of derivation

```
class TextParagraph : public Thing {
    …
};

class CircuitThing : public Thing {
    …
};
```

Public derivation acknowledges that both `TextParagraph` and `CircuitThing` are specialized kinds of `Things` and so code "using `Things`" will work with `TextParagraphs` or `CircuitThings`. This is exactly what we want for the example where the `Document` object has a list of "pointers to `Things`" and all its code is of the form `thing_ptr->DoSomething()`.

public derivation

We need actual `TextParagraph` objects. This class has to be "concrete". The class declaration has to be complete, and all the member functions will have to be defined.

TextParagraph, a concrete class

Naturally, the class declaration starts with the constructor(s) and destructor. Then it will have to repeat the declarations from class `Thing`; so we again get functions like `Draw()` being declared. This time they don't have those `= 0` definitions. There will have to be definitions provided for each of the functions. (It is not actually necessary to repeat the keyword `virtual`; this keyword need only appear in the class that introduces the member function. However, it is usually simplest just to "copy and paste" the block of function declarations and so have the keyword.) Class `TextParagraph` will introduce some additional member functions describing those behaviours that are unique to `TextParagraph`s. Some of these additional functions will be in the public interface; most would be private. Class `TextParagraph` would also declare all the private data members needed to record the data possessed by a `TextParagraph` object.

CircuitThing, a partially implemented abstract class

Class `CircuitThing` is an in between case. It is not a pure abstraction like `Thing`, nor yet is it a concrete class like `TextParagraph`. Its main role is to introduce those member functions needed to specify the behaviours of all different kinds of `CircuitThing` and to describe those data members that are possessed by all kinds of `CircuitThing`.

Class `CircuitThing` cannot provide definitions for all of those pure virtual functions inherited from class `Thing`; for instance it can't do much about `Draw()`. It should not repeat the declarations of those functions for which it can't give a definition. Virtual functions only get re-declared in those subclasses where they are finally defined.

Class `CircuitThing` can specify some of the processing that must be done when a `CircuitThing` gets written to or read from a file on disk. Obviously, it cannot specify everything; each specialized subclass has its own data to save. But `CircuitThing` can define how to deal with the common data like the identifier, location and label:

```
void CircuitThing::WriteTo(ostream& os) const
{   // keyword virtual not repeated in definition
    os << fId << endl;
    os << fLocation.x << " " << fLocation.y << endl;
    os << fLabel << endl;
}

void CircuitThing::ReadFrom(istream& is)
{
    is >> fId;
    is >> fLocation.x >> fLocation.y;
    char buff[256];
    is.getline(buff,255,'\n');
    delete [] fLabel; // get rid of existing label
    fLabel = new char[strlen(buff) + 1];
    strcpy(fLabel, buff);
}
```

These member functions can be used by the more elaborate `WriteTo()` and `ReadFrom()` functions that will get defined in subclasses. (Note the deletion of `fLabel`

and allocation of a new array; this is another of those places where it is easy to get a memory leak.)

The example illustrates that there are three possibilities for additional member functions:

```
int GetId() const { return this->fId }
virtual Rect      Area() const {
    return Rect(
           this->flocation.x - 8, this->flocation.y - 8,
           this->flocation.x + 8, this->flocation.y + 8);
           }
virtual double Current() const = 0;
```

A non-virtual member function

Function `GetId()` is not a virtual function. Class `CircuitThing` defines an implementation (return the `fId` identifier field). Because the function is not virtual, subclasses of `CircuitThing` cannot change this implementation. You use this style when you know that there is only one reasonable implementation. for a member function.

A defined, virtual member function

Function `Area()` has a definition. It creates a rectangle of size 16x16 centred around the `fLocation` point that defines the centre of a `CircuitThing`. This might suit most specialized kinds of `CircuitThing`; so, to economise on coding, this default implementation can be defined at this level in the hierarchy. Of course, `Area()` is still a virtual function because that was how it was specified when first introduced in class `Thing` ("Once a virtual function, always a virtual function"). Some subclasses, e.g. class `Wire`, might need different definitions of `Area()`; they can override this default definition by providing their own replacement.

Another pure virtual function

Function `Current()` is an additional pure virtual function. The circuit simulation code will require all circuit elements know the current that they have flowing. But the way this gets calculated would be class specific.

Access to members

Class `CircuitThing` declares some of the data members – `fId`, `fLabel`, and `fLocation`. There is a potential difficulty with these data members.

These data members should not be `public`; you don't want the data being accessed from anywhere in the program. But if the data members are declared as `private`, they really are private, they will only be accessible from the code of class `CircuitThing` itself. But you can see that the various specialized subclasses are going to have legitimate reasons for wanting to use these variables. For example, all the different versions of `Draw()` are going to need to know where the object is located in order to do the correct drawing operations.

You can't use the "`friend`" mechanism to partially relax the security. When you define class `CircuitThing` you won't generally know what the subclasses will be so you can't nominate them as friends.

"protected" data

There has to be a mechanism to prevent external access but allow access by subclasses– so there is. There is a third level of security on members. In addition to `public` and `private`, you can declare data members and member functions as being

protected. A protected member is not accessible from the main program code but can be accessed in the member functions of the class declaring that member, or in the member functions of any derived subclass.

Here, variables like fLocation should be defined as protected. Subclasses can then use the fLocation data in their Draw() and other functions. (Actually, it is sometimes better to keep the data members private and provide extra protected access functions that allow subclasses to get and set the values of these data members. This technique can help when debugging complex programs involving elaborate class hierarchies).

Once the definition of class CircuitThing is complete, you have to continue with its derived classes: class Wire, and class Component:

```
class Wire : public CircuitThing {
public:
    Wire(int startcomponent, int endcomponent, Point p1, Point p2);
    ~Wire();
    /* Disk I/O */                                    Thing declared behaviours
    virtual void  ReadFrom(istream& is);
    virtual void  WriteTo(ostream& os) const;
    /* Graphics */
    virtual void  Draw() const;
    /* mouse interactions */
    virtual void  DoDialog();   // For double click
    virtual void  TrackMouse(); // Mouse select and drag
    virtual Rect  Area() const;
    virtual double Current() const;                   CircuitThing behaviours
    …
    int    FirstEndId() { return this->fFirstEnd; }   Own unique behaviours
    …
private:
    int    fFirstEnd;
    …
};
```

Class Wire is meant to be a concrete class; the program will use instances of this class. So it has to define all member functions.

The class repeats the declarations for all those virtual functions, declared in classes from which it is derived, for which it wants to provide definitions (or to change existing definitions). Thus class Wire will declare the functions like Draw() and Current(). Class Wire also declares the ReadFrom() and WriteTo() functions as these have to be redefined to accommodate additional data, and it declares Area() as it wants to use a different size.

Class Wire would also define additional member functions characterising its unique behaviours and would add some data members. The extra data members might be declared as private or protected. You would declare them as private if you knew that no-one was ever going to try to invent subclasses based on your class Wire. If you wanted to allow for the possibility of specialized kinds of Wire, you would make these

extra data members (and functions) `protected`. You would then also have to define the destructor as `virtual`.

The specification of the problem might disallow the user from dragging a wire or clicking on a wire to open a dialog box. This would be easily dealt with by making the `Area()` function of a `Wire` return a zero sized rectangle (rather than the fixed 16x16 rectangle used by other `CircuitThings`):

```
Rect Wire::Area() const
{
    return Rect(0, 0, 0, 0);
}
```

(The program identifies the `Thing` being selected by testing whether the mouse was located in the `Thing`'s area; so if a `Thing`'s area is zero, it can never be selected.) This definition of `Area()` overrides that provided by `CircuitThing`.

A `Wire` has to save all the standard `CircuitThing` data to file, and then save its extra data. This can be done by having a `Wire::WriteTo()` function that makes use of the inherited function:

```
void Wire::WriteTo(ostream& os)
{
    CircuitThing::WriteTo(os);
    os << fFirstEnd << " " << fSecondEnd << endl;
    …
}
```

This provides another illustration of how inheritance structures may lead to small savings of code. All the specialized subclasses of `CircuitThing` use its code to save the identifier, label, and location.

23.6.3 BUT HOW DOES IT WORK?!

The example hierarchy illustrates that you can define a concept like `Thing` that can save itself to disk, and you can define many different specific classes derived from `Thing` that have well defined implementations – `TextParagraph::WriteTo()`, `Battery::WriteTo()`, `Wire::WriteTo()`. But the code for `Document` would be something like:

```
void Document::DoSave(ostream& out)
{
    out << thingList.Length() << endl;

    iterator i1(thingList);
    i1.First();
    while(!i1.IsDone()) {
           Thing* thing_ptr = (Thing*) i1.CurrentItem();
```

```
            thing_ptr ->WriteTo(out);
            i1.Next();
            }
    }
```

The code generated for

```
thing_ptr ->WriteTo()
```

isn't supposed to invoke function `Thing::WriteTo()`. After all, this function doesn't exist (it was defined as `= 0`). Instead the code is supposed to invoke the appropriate specialized version of `WriteTo()`.

But which is the appropriate function? That is going to depend on the contents of `thingList`. The `thingList` will contain pointers to instances of class `TextParagraph`, class `Battery`, class `Switch` and so forth. These will be all mixed together in whatever order the user happened to have added them to the `Document`. So the appropriate function might be `Battery::WriteTo()` for the first `Thing` in the list, `Resistor::WriteTo()` for the second list element, and `Wire::WriteTo()` for the third. You can't know until you are writing the list at run-time.

The compiler can't work things out at compile time and generate the instruction sequence for a normal subroutine call. Instead, it has to generate code that works out the correct routine to use at run time.

virtual tables

The generated code makes use of tables that contain the addresses of functions. There is a table for each class that uses `virtual` functions; a class's table contains the addresses of its (virtual) member functions. The table for class `Wire` would, for example, contain pointers to the locations in the code segment of each of the functions `Wire::ReadFrom()`, `Wire::WriteTo()`, `Wire::Draw()` and so forth. Similarly, the virtual table for class `Battery` will have the addresses of the functions `Battery::ReadFrom()` and so on. (These tables are known as "virtual tables".)

In addition to its declared data members, an object that is an instance of a class that uses `virtual` functions will have an extra pointer data member. This pointer data member holds the address of the virtual table that has the addresses of the functions that are to be used in association with that object. Thus every `Wire` object has a pointer to the `Wire` virtual table, and every `Battery` object has a pointer to the `Battery` virtual table. A simple version of the scheme is illustrated in Figure 23.6

The instruction sequence generated for something like:

```
thing_ptr ->WriteTo()
```

involves first using the link from the object pointed to by `thing_ptr` to get the location of the table describing the functions. Then, the required function, `WriteTo()`, is "looked up" in this table to find where it is in memory. Finally, a subroutine call is made to the actual `WriteTo()` function. Although it may sound complex, the process requires only three or four instructions!

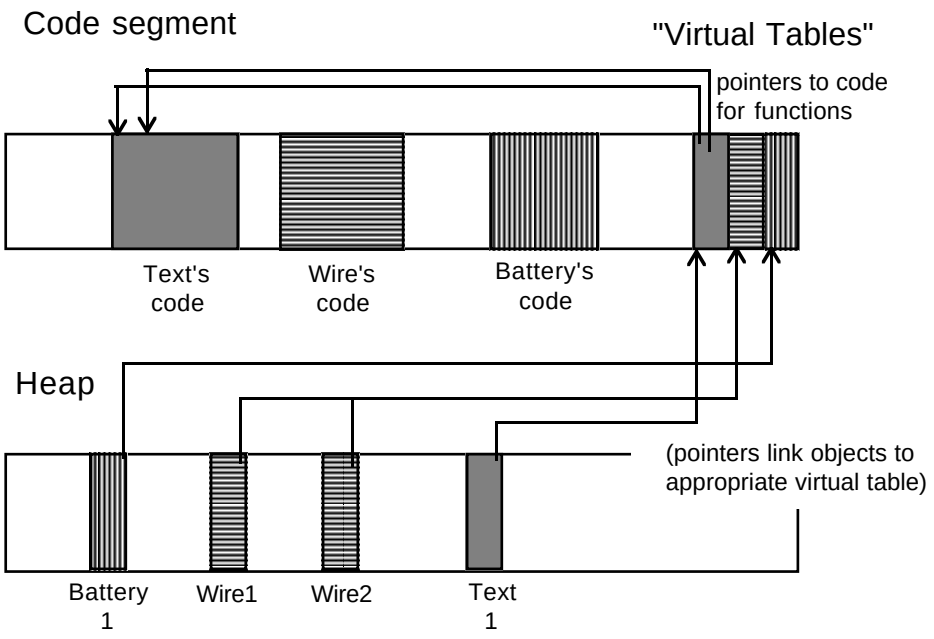

Figure 23.6 Virtual tables.

Dynamic binding

Function lookup at run time is referred to as "dynamic binding". The address of the function that is to be called is determined ("bound") while the program is running (hence "dynamically"). Normal function calls just use the machine's `JSR` (jump to subroutine) instruction with the function's address filled in by the compiler or linking loader. Since this is done before the program is running, the normal mechanism of fixing addresses for subroutine calls is said to use static binding (the address is fixed, bound, before the program is moving, or while it is static).

It is this "dynamic binding" that makes possible the simplification of program design. Things like `Document` don't have to have code to handle each special case. Instead the code for `Document` is general, but the effect achieved is to invoke different special case functions as required.

Polymorphism

Another term that you will find used in relation to these programming styles is "polymorphism". This is just an anglicisation of two Greek words – poly meaning many, and morph meaning shape. A `Document` owns a list of `Things`; `Things` have many different shapes – some are text paragraphs, others are wires. A pointer like `thing_ptr` is a "polymorphic" pointer in that the thing it points to may, at different times, have different shapes.

23.6.4 MULTIPLE INHERITANCE

You are not limited to single inheritance. A class can be derived from a number of existing base classes.

Multiple inheritance introduces all sorts of complexities. Most uses of multiple inheritance are inappropriate for beginners. There is only one form usage that you should even consider.

Multiple inheritance can be used as a "type composition" device. This is just a systematic generalization of the previous example where we had class `Thing` that represented the type "a drawable, storable, editable data item occupying an area of a window".

Instead of having class `Thing` as a base class with *all* these properties, we could instead factor them into separate classes:

```
class Storable {
public:
    virtual ~Storable() { }
    virtual void WriteTo(ostream&) const = 0;
    virtual void ReadFrom(istream&) const = 0;
    …
};

void Drawable {
public:
    virtual ~Drawable() { }
    virtual void Draw() const = 0;
    virtual Rect Area() const = 0;
    …
};
```

This allows "mix and match". Different specialized subclasses can derive from chosen base classes. As a `TextParagraph` is to be both storable and drawable, it can inherit from both base classes:

```
class TextParagraph : public Storable, public Drawable {
…
};
```

You might have another class, `Decoration`, that provides some pretty outline or shadow effect for a drawable item. You don't want to store `Decoration` objects in a file, they only get used while the program is running. So, the `Decoration` class only inherits from `Drawable`:

```
class Decoration : public Drawable {
…
};
```

As additional examples, consider class `Printable` and class `Comparable`:

```
class Printable {
public:
    virtual ~Printable() { }
    virtual void PrintOn(ostream& out) const = 0;
};

ostream& operator<<(ostream& o, const Printable& p)
{ p.PrintOn(o); return o; }
ostream& operator<<(ostream& o, const Printable *p_ptr)
{ p_ptr->PrintOn(o); return o; }

class Comparable {
public:
    virtual ~Comparable() { }
    virtual int Compare(const Comparable* ptr) const = 0;
    int Compare(const Comparable& other) const
           { return Compare(&other); }

    int operator==(const Comparable& other) const
           { return Compare(other) == 0; }
    int operator!=(const Comparable& other) const
           { return Compare(other) != 0; }
    int operator<(const Comparable& other) const
           { return  Compare(other) < 0; }
    int operator<=(const Comparable& other) const
           { return  Compare(other) <= 0; }
    int operator>(const Comparable& other) const
           { return  Compare(other) > 0; }
    int operator>=(const Comparable& other) const
           { return  Compare(other) >= 0; }
};
```

Class `Printable` packages the idea of a class with a `PrintOn()` function and associated global `operator<<()` functions. Class `Comparable` characterizes data items that compare themselves with similar data items. It declares a `Compare()` function that is a little like `strcmp()`; it should return -1 if the first item is smaller than the second, zero if they are equal, and 1 if the first is greater. The class also defines a set of operator functions, like the "not equals function" operator `!=()` and the "greater than" function `operator>()`; all involve calls to the pure virtual `Compare()` function with suitable tests on the result code. (The next chapter has some example `Compare()` functions.)

As noted earlier, another possible pure virtual base class would be class `Iterator`:

```
class Iterator {
public:
    virtual ~Iterator() { }
    virtual void  First(void) = 0;
```

```
    virtual void  Next(void) = 0;
    virtual int   IsDone(void) const  = 0;
    virtual void  *CurrentItem(void) const  = 0;
};
```

This would allow the creation of a hierarchy of iterator classes for different kinds of data collection. Each would inherit from class `Iterator`.

Now inventing classes like `Storable`, `Comparable`, and `Drawable` is not a task for beginners. You need lots of experience before you can identify widely useful abstract concepts like the concept of storability. However you may get to work with library code that has such general abstractions defined and so you may want to define classes using multiple inheritance to combine different data types.

What do you gain from such use of inheritance as a type composition device?

Obviously, it doesn't save you any coding effort. The abstract classes from which you multiply inherit are exactly that – abstract. They have no data members. All, or most, of their member functions are pure virtual functions with no definitions. If any member functions are defined, then as in the case of class `Comparable`, these definitions simply provide alternative interfaces to one of the pure virtual functions.

You inherit, but the inheritance is empty. You have to define the code.

The advantage is not for the implementor of a subclass. Those who benefit are the maintenance programmers and the designers of the overall system. They gain because if a project uses such abstract classes, the code becomes more consistent, and easier to understand. The maintenance programmer knows that any class whose instances are to be stored to file will use the standard functions `ReadFrom()` and `WriteTo()`. The designer may be able to simplify the design by using collections of different kinds of objects as was done with the `Document` example.

23.6.5 USING INHERITANCE

There are many further complexities related to inheritance structures. One day you may learn of things like "private inheritance", "virtual base classes", "dominance" and others. You will discover what happens if a subclass tries to "override" a function that was not declared as `virtual` in the class that initially declared it.

But these are all advanced, difficult features.

The important uses of inheritance are those illustrated – capturing commonalities to simplify design, and using (multiple) inheritance as a type composition device. These uses will be illustrated in later examples. Most of Part V of this text is devoted to simple uses of inheritance.