Chapter 5

The Class Definition

The most obvious area of similarity between Java and C++ is in the structure of a class definition. However, although there are obvious superficial similarities between the two, there are behind these a host of both syntactic and subtle semantic differences. In this chapter, and in subsequent chapters, we will explore some of these differences.

5.1 Obvious Similarities

A class definition in Java (Figure 5.1) is similar to a class definition in C++ (Figure 5.2) in that they both begin with the keyword class, and they are both demarcated by a pair of curly braces.

Then there are differences. Some differences are reasonably minor. Class definitions in C++ are statements, and must end with a semicolon. Class definitions in Java have no semicolon. Class definitions in C++ are divided into major sections by the keywords private, protected and public, which are sectioning commands followed by a colon. Java applies these modifiers to each data field or method individually, omitting the colon. Java uses the keyword extends to indicate inheritance from a parent class, C++ uses a colon and the

```
class box {

    public box (int v) { val = v; }
    public int value() { return val; }

    private int val;
}
```

Figure 5.1: A Typical Class Definition in Java

```
class box {
public:
    box (int v) { val = v; }
    int value() { return val; }

private:
    int val;
};
```

Figure 5.2: A Typical Class Definition in C++

keyword public. (In Chapter 6 we will point out that the keywords private or protected could also be used here, although the meaning is then slightly different). Java permits the modifier public to be applied to an entire class, whereas all classes in C++ have this attribute. (The more important difference is between classes that are *not* declared as public in Java. The closest C++ equivalent to these is produced using a non-public namespace, a relatively recent addition to the C++ language. See Section 12.7 for a discussion of namespaces.)

Once past the minor differences, we enter the realm of more substantive issues. The following sections will explore many of these.

5.2 Separation of Class and Implementation

A class definition in Java is a single unit, and everything related to the class is found between the opening and closing curly brace. In C++, on the other hand, the class definition is only the starting point, defining only the structure and interface for the class. Many features associated with the class will be found outside the class definition, sometimes even outside the file in which the class definition appears.

In Java all methods are defined within the class itself. In C++ this is true only for the smallest methods. Furthermore, there is a semantic difference between methods defined within a class and those not.

DEFINE *Invoking an inline method does not generate a function call, but rather expands the body at the point of call*

A method that is provided with a definition as part of the class body is called an *inline* method. Such a method may (at the discretion of the C++ compiler) be expanded in-line at the point of call. That is, when the method function is invoked, rather than issuing a function call instruction, the compiler may elect to expand the body of the method directly into the code being produced for the caller.

RULE *Only use in-line definitions for methods that are short*

The advantage of expanding a method in-line is one of speed, as such a function avoids the overhead of a procedure call statement. This advantage is purchased at a price of space and complexity. The code for the caller may become larger, and the code is (internally to the compiler at least, if not for the programmer) more complex. For this reason, in-line definitions should only be used for short method bodies. A good rule of thumb is to only use in-line definitions for assignment statements and return statements, or at most a single conditional statement. Never use an in-line definition for a code fragment that contains a loop or a recursive function call.

When methods are not defined in-line, they must be provided with a definition elsewhere. Since methods in different classes may have the same name, the method definition must be tied to the appropriate class. This is accomplished by defining a method using a *fully qualified* function name. The following illustrates this:

NOTE *Method definitions in C++ can be separate from the class definition*

```
void Link::addBefore (int val, List * theList)
{
        // create a new link
    Link * newLink = new Link(val, this, backwardLink);
        // put it into the appropriate place
    if (backwardLink == 0)     // replacing first element in list
        theList->firstLink = newLink;
    else {     // inserting into the middle of the list
        backwardLink->forwardLink = newLink;
        backwardLink = newLink;
        }
}
```

The double colon indicates that full qualification is being employed. The prefix indicates the class (Link, in this example) while the text after the double colon indicates the function name (addBefore). This method definition appears outside the class definition itself. The qualified name can be thought of as similar to a persons full name; the name Tom Smith identifies a unique individual better than simply the first name Tom, which may have many different associations.

5.2.1 Interface and Implementation Files

A further form of separation in C++ involves the division of a program into multiple files. A class that is used in many files is normally described in an *interface* file. Traditionally the extension .h is used in the interface file name. The interface file describes just the bare bones information concerning the services provided by a class, but not how the class goes about implementing that behavior. Method bodies are found in a separate file, called the *implementation* file. Traditionally implementation files are named by file names ending in .cpp or .c++.

NOTE *Implementations will often be in a different file than the class definition*

C++ is much less concerned than Java with the relationship between class and file names. It is not necessary that a class be defined in a file of the same name, although adopting this as a convention certainly makes file management a little bit easier.

WARNING *In C++ a class need not be defined in a file with the same name*

Every file that must use a class will include the appropriate interface file, using a statement such as the following:

```
# include <libClass.h>
# include "myClass.h"
```

The brackets surrounding the file name are used to indicate where the requested interface file is to be found. Angle brackets indicate "system" interface files, those that are provided as part of standard libraries, and are found in the designated location for such libraries. Quote marks are used for immediate interface files, those that are found in the same directory as the program being compiled.

The include statement differs from the import statement in Java, in that the include statement performs a textual inclusion of the given file at the point where the include statement is written. Note also that there is no semicolon following an include statement, while a semicolon is necessary following an import statement in Java.

5.2.2 The inline directive

Normally class definitions are found in an interface file, and methods are found in an implementation file. For classes an exception is made if a class is used in only one file, when it can simply be written in the implementation file. Similarly, method implementations will occasionally be written in an interface file. When they are, an inline directive can be used to indicate that the function can be expanded in-line at a point of call, exactly as if the method had been written in a class description.

Inline methods are most often necessary to overcome the more stringent naming restrictions in C++. Names in general in C++ must be known to the compiler at the first point they are used. In Java, on the other hand, names can often be defined subsequent to their first use. Often a method may need to access another feature that is not known at the time the class definition is processed, but is known shortly thereafter. In such cases the definition of the name is given between the class definition and the description of the method body. If the method body is sufficiently short, it can then be marked as inline. We will see an example of this later in this chapter.

Template methods (see Chapter 9) are also normally found in an interface file, regardless of their size.

Virtual methods (see Chapter 4) should not be declared as inline, as the compiler is not able to produce in-line code even if requested by the user.

5.2.3 Prototypes

Another common feature found in an interface file is a list of function prototypes. In order to perform strong type checking, function type signatures must be known before a function can be invoked. A *function type signature* is a description of the types of all the arguments, as well as the return type for a function.

DEFINE
A type signature describes the argument and return types of a function

A function prototype is simply a function heading, with the function body and, optionally, the argument names removed. Thus, the only information conveyed by the prototype is the function name, the argument types and the result type. The following are some examples:

```
int max (int, int);
int min (int a, int b); // argument names are optional
complex abs (complex &); // can use user defined types
bool operator < (distance &, distance &); // prototype for operator
```

Some C++ compilers will issue warnings if a function is defined without a prior prototype directive.

A curious holdover from the C world is occasionally encountered. In earlier versions of C a prototype with an empty argument list, such as the following:

```
int fun ();
```

only indicated the existence of a function by the given name. No information concerning the arguments, if any, was implied. Instead, to indicate that the function took no arguments, it was necessary to explicitly indicate the fact, using the keyword void:

```
int fun (void); // no arguments, returns an int
```

The language C++ recognizes both declarations, but unlike C assumes that if no arguments are given, none are required.

5.2.4 External Declarations

The extern modifier to a declaration indicates that a global variable is defined in another file, but will be used in the current file:

NOTE *A variable that is used in two or more files must be declared external*

```
extern int size;
extern char buffer[ ]; // array limits don't have to be given
extern ObjectType anObject;
```

The declaration informs the linker that the value being named is used in two or more files, but should nevertheless refer to only one object. Such declarations are most generally found in header files, although there is no restriction on their appearing in implementation files.

Another form of the statement is used to indicate that a C++ program is being linked with a function written in a different language.

```
    // declare strcmp is written in C, not C++
extern "C" int strcmp (const char *, const char *);
```

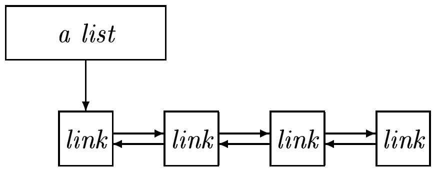
5.3 Forward References

NOTE *Both function and class names must be defined before they can be used*

The C++ language resolves names at the point they are used. This means that when a class name is used, say in a declaration, the name must already be known to be a class. Similarly when a function is invoked, a declaration (in prototype at least) for the function must already have been seen. In Java, on the other hand, references are resolved only after an entire file has been processed.

Prototypes are used to get around the problem of function definitions. For classes the language permits a *forward declaration*. A forward declaration asserts that a particular name represents a class, but gives no further information. Such a declaration permits pointers to be declared to the class, but not invoke methods defined by the class nor the creation of instances of the class.

To illustrate, assume we wish to implement a linked list abstraction, using two classes, List and Link. Instances of List will simply reference the first Link in the collection. Thereafter each link will point to the next, and to the previous:



The class List must include a pointer to a Link, however the class Link might also want to reference features in List. In the example we illustrate this possibility by writing the method addBefore, which adds a new value immediately before a given link. The code for this method was presented earlier. The solution to the referencing problem is to first provide a forward declaration for the Link class before the definition of List. The forward definition is sufficient to permit the declaration of a data field holding a pointer to an object of type Link, although not the execution of any methods associated with the class:

```
class Link;     // forward declaration

class List {
public:
    ...
private:
    Link * firstLink; // permitted, since class Link is declared
```

```
    void push_front (int val);
};

class Link {     // now provide the full link implementation
public:
        // data fields are public
    int value;
    Link * forwardLink;
    Link * backwardLink;

        // constructor defined in-line
    Link (int v, Link * f, Link * b)
    {
        value = v;
        forwardLink = f;
        backwardLink = b;
    }

        // prototype, definition given elsewhere
        // requires knowledge of class List
    void addBefore (int val, List * theList);
};
```

This situation, where we have a pair of mutually recursive classes, is a common places where an explicitly inline method definition is appropriate. For example, the method push_front in class List is a suitable candidate for inlining, but can only be written once the class definition for Link has been seen, since it uses methods from that class:

```
inline void List::push_front (int val)
{
    if (firstElement == 0)    // adding to empty list
        firstElement = new Link(val, 0, 0);
    else         // else add before first element
        firstElement->addBefore(val, this);
}
```

5.4 Constructors and Initialization

Constructors in Java and in C++ are designed to serve the same purpose, namely to tie together the two tasks of creation and initialization, thereby ensuring that no value is

DEFINE *Constructors tie together the tasks of creation and initialization* created without being initialized, and that no value is initialized more than once. In both languages a constructor is written as a method that shares the same name as a class. Beyond these similarities, there are a number of important differences.

5.4.1 Default and Copy constructors

In C++ two types of constructors are used not only for explicit initialization associated with a declaration statement, but also implicitly for a variety of situations that occur during the course of execution. These two constructor patterns are given special names.

RULE *All class definitions should include both a default and copy constructor*

A *default constructor* is a constructor that does not take any arguments. The constructor is invoked with a declaration of an object values does not specify argument values. It is also used to initialize object data fields, when no other arguments are specified.

A *copy constructor* is a constructor that takes an instance of the same class as a constant reference argument. Copy constructors are used to make a copy, or clone, of an object value, a task that in Java is often performed using the clone method. Copy constructors are used internally in the processing of parameters that are passed by-value. The copy constructor will be invoked to create the temporary value that will be passed to the function, leaving the original value immune to modification. Copy constructors are also used to create a temporary value when a function returns an object value, and (as are all constructors) in the initialization of values newly created by either a declaration or a new operation.

The following class description illustrates both a default and a copy constructor:

```
class box {
public:
    box ()          // default constructor
        { i = 1; } // give data field some default value

    box (int x)     // ordinary constructor
        { i = x; }

    box (const box & a)     // copy constructor
        { i = a.i; } // clone argument value
private:
    int i;
};
```

The following three declarations will implicitly invoke the three forms of constructor:

```
box one;        // default constructor
box two(7);     // ordinary constructor
box three(two); // copy constructor
```

5.4.2 Initializers

Data members in Java are initialized in one of two ways. If a data member is initialized with a value that is independent of the constructor arguments, it is often simply written as an initial assignment at the point of declaration. Otherwise, an explicit assignment statement appears in the constructor function definition.

With one minor exception (to be discussed in Section 5.8) the C++ language does not allow the initialization of data members at the point of declaration. Instead, all data members must be initialized in a constructor function. This can be performed either in an explicit assignment statement, or in an *initializer*. The following class definition illustrates the syntax used for initializers, by rewriting the Link constructor presented earlier to use initializers, rather than assignment statements:

```
class Link {
public:
    Link(int v, Link * f, Link * b)
        : value(v), forwardLink(f), backwardLink(b) { }
    ...
};
```

For primitive data types, such as integers or pointers, an initializer is exactly equivalent to an assignment. For more complex types, such as user defined types, the situation is different. The rule is that data members are all initialized before the body of the constructor is executed, either by initializer or, if no initializers are known, using the default rules.

Imagine a class box that defines both a default constructor (a constructor with no argument), a copy constructor (a constructor that creates a clone of another value), and an assignment operation. Now consider the following class definition:

```
class A { // class with initialization error
public:
    void A (box & aBox) : boxOne(aBox) { boxTwo = aBox; }
private:
    box boxOne;
    box boxTwo;
};
```

RULE
Use initializers when possible

The data field boxOne will be initialized using the copy constructor, in one step. The data field boxTwo, on the other hand, has no initializer field, and will thus first be initialized using the default constructor. After both the data members have been initialized, the body of the constructor is executed. In there, the assignment statement is used to alter the value of boxTwo to match the argument value. Thus, the field boxTwo is modified twice, once by the default constructor and once by the assignment operator.

Data members that are declared to be const and data fields that are references are never permitted to be targets of assignment statements. Thus both of these types of objects must be initialized using initializers, instead of in the body of the constructor.

```
class B {
public:
    void B (box & aBox) : boxOne(aBox), boxTwo(aBox) { }
private:
    box & boxOne;
    const box boxTwo;
}
```

NOTE *The language C++ does not use the keyword* super

The last category of initializer is the initialization of parent classes in the constructors associated with child classes. Assume that the constructor for a parent class requires an argument value. In Java, the constructor for the child supplies the value by invoking the function super in the constructor:

```
class bigBox extends box { // Java code

    public bigBox (int x, double d)
    {
        super(x); // initialize parent
        dvalue = d; // initialize self
    }

    private double dvalue; // private data field
}
```

In C++ the same effect is achieved by means of an initializer. The initializer names the parent class, and uses as arguments the arguments for the parent class constructor. As in Java, if no appropriate initializer is found the default constructor for the parent class is invoked.

```
class bigBox : public box { // C++ code
public:
    bigBox (int x, double d) : box(x), dvalue(d) { }

private:
    double dvalue;
};
```

NOTE *Fields are initialized in the order of declaration*

A final item to note is that class members are initialized in the order that they are declared in the class body, not in the order the initializers are listed. (They are deleted by the *destructor* in the reverse order. Destructor functions were introduced in Chapter 4.) The programmer might be very confused by the result of executing the method test on an instance of the following class:

```
class order {  // warning, initialization error
public:
    order (int i) : one(i), two(one) { }
    int test() { return two; }

private:
    int two; // initialized first
    int one; // initialized second
};
```

The variable two is initialized with the as yet uninitialized value of one, and *then* the value of one is set. The result is that the value held in two is unpredictable garbage.

5.4.3 Order of Initialization

WARNING
C++ and Java initialize subclasses differently

In C++ the initialization of parent classes occurs before the initialization of child classes. During the time a parent class constructor is executed the object is viewed as an instance of the parent class. This means that methods that are invoked are matched only to functions in the parent class, even if these methods have been declared as *virtual*. (See Chapter 4 for a discussion of the *virtual* modifier). To see the impact of this, we can once again contrast a simple class written in Java and the equivalent class written in C++. Consider first the following Java classes:

```
class A { // Java classes illustrating initialization
    public A ()
    {
        System.out.println("in A constructor");
        init();
    }

    public void init()
    {
        System.out.println("in A init");
    }
}

class B extends A {
```

```
    public B ()
    {
        System.out.println("in B constructor");
    }

    public void init()
    {
        super.init();
        System.out.println("in B init");
    }
}
```

DEFINE *A refinement combines the actions of the parent and child*

When an instance of B is created the constructor for B will be executed. This constructor will automatically invoke the constructor for A. The constructor for A will invoke the method init, which is defined in A but overridden in B. However, the overridden method in B will invoke the method in A. (As an aside, notice that constructors always use *refinement* semantics, in which the parent class will always be invoked. Methods, on the other hand, use *replacement* semantics, and so the parent function is only invoked if explicitly called). The output of the sequence would be as follows:

```
in A constructor
in A init
in B init
in B constructor
```

A superficially equivalent C++ program is as follows:

```
class A { // C++ classes illustrating initialization
public:
    A ()
    {
        printf("in A constructor\n");
        init();
    }

    virtual void init()
    {
        printf("in A init\n");
    }
};

class B : public A {
```

```
public:

    B ()
    {
        printf("in B constructor\n");
    }

    virtual void init()
    {
        A::init();
        printf("in B init\n");
    }
};
```

However, in C++ when the function init is invoked in the constructor for A, only the method in class A is used, regardless whether or not the virtual keyword is used in the declaration of the function. Thus, the output from C++ would be as follows:

```
in A constructor
in A init
in B constructor
```

Notice that the init function in B has not been executed at all. Should the C++ programmer attempt to overcome this limitation by invoking the init function directly in B, they would discover another error, namely that the init function in A would then be invoked *twice*.

5.4.4 Combining Constructors

Java programmers are accustomed to being able to define one constructor using another, as in the following:

```
    // Java class with linked constructors
class newClass {
    public newClass (int i)
    {
        // do some initialization
        ...
    }

    public newClass (int i, int j)
    {
```

```
        this(i); // invoke one argument constructor
        // do other initialization
        ...
    }
}
```

WARNING
In C++ you cannot invoke a constructor from within another

This feature has no direct C++ counterpart. Oftentimes programmers will try to use a fully qualified name, as in the following:

```
class box {     // error - does not work as expected
public:
    box (int i) : x(i) { }
    box (int i, int j) : y(j) { box::box(i); }

    int x, y;
};
```

Creating a two-argument box and printing the resulting value of x and y will demonstrate that the fully qualified call on the one-argument constructor had no effect. In reality, what this function accomplished was to create an unnamed temporary, initialize it, then destroy it.

There are two common solutions that demonstrate the proper solution to this problem. If the only difference between two constructors is the assignment of a default value, then a *default argument value* can be used:

```
    // C++ class with default arguments in constructor
class newClass {
public:
    newclass (int i, int j = 7)
    {
        // do object initialization
        ...
    }
};
```

Although only one function is defined, it can be used with either one argument or two. In the one argument case, the default value (here seven) is automatically supplied for the second argument.

Another common solution to this problem is to factor the common initialization into a separate method, which is then declared as private:

```
    // C++ class with factored constructors
```

```
class newClass {
public:
    newClass (int i)
    {
        initialize(i); // do common initialization
    }

    newClass (int i, int j)
    {
        initialize(i);
        ... // then do further initialization
    }

private:
    void initialize (int i)
    {
        ... // common initialization actions
    }
};
```

Even if the initialize method is declared public or protected as well as virtual, it cannot be overridden since it is invoked from within a constructor function. (See Section 5.4.3).

5.5 The Orthodox Canonical Class Form

Several authors of style guides for C++ have suggested that almost all classes should define four important functions. This has come to be termed the *orthodox canonical class* form. The four important functions are:

RULE *Always define the four functions in the OCCF*

- A default constructor. This is used internally to initialize objects and data members when no other value is available.
- A copy constructor. This is used in the implementation of call-by-value parameters.
- An assignment operator. This is used to assign one value to another.
- A destructor. This is invoked when an object is deleted.

Even if empty bodies are supplied for these functions, writing the class body will at least suggest that the program designer has *thought* about the issues involved in each of these. Furthermore, appropriate use of visibility modifers, described in the next section, give the programmer great power is allowing or disallowing different operations used with the class.

5.6 Visibility modifiers

For the most part, the visibility modifiers public, protected, and private operate the same in C++ as they do in Java. However, their are some differences. In C++ the modifiers designate a section of a class definition, rather than being applied item by item as they are in Java. The modifiers cannot be applied to entire classes in C++. One way to get the effect of a non-public class in C++ is to declare a class inside a *namespace* (see Section 12.7).

There is a minor difference in the meaning of the keyword protected. Protected fields in Java are open to both subclasses and to other classes declared within the same package. The C++ language has no notion of packages, and so the term applies only to subclasses.

WARNING
A subclass is permitted to change the visibility of attributes inherited from a parent class

The C++ language allows a subclass to change the visibility of a method, even one that is declared as virtual and is being overridden. Consider the following:

```
class parent {
public:
    virtual void test () { printf("in parent test\n"); }
};

class child : public parent {
private:
    void test () { printf("in parent test\n"); }
};

parent * p = new child;
p->test();
```

A variable of type pointer to parent is nevertheless holding a value of type child. The method test is defined in both classes. When invoked, the method executed will be that found in the child class, not that found in the parent class. That much, except for the funny keyword virtual that we will discuss later in Chapter 6, should come as no surprise to the Java programmer.

What *is* surprising is that the method test was executed, despite the fact that it was declared as private in the class definition. Such a method cannot, after all, be invoked from a variable declared as child, even if it holds the exact same value:

```
child * c = (parent *) p;
c->test(); // compile error, cannot invoke private method
```

Some compilers will produce warning messages when the visibility of methods is changed in this fashion, but unlike Java it is not considered a compiler error.

5.7 Inner classes versus Nested Classes

WARNING
C++ and Java both allow classes to be defined inside each other, but have different semantics

Both Java and C++ permit class definitions to be nested. A class definition that appears inside another class definition is termed an *inner class* in Java, and a *nested class* in C++. Despite the similar appearances, there is a major semantic difference between the two concepts. An inner class in Java is linked to a specific instance of the surrounding class (the instance in which it was created), and is permitted access to data fields and methods in this object. A nested class in C++ is simply a naming device, it restricts the visibility of features associated with the inner class, but otherwise the two are not related.

To illustrate the use of nested classes let us rewrite the Java linked list abstraction presented earlier, placing the Link class inside the List abstraction:

```
    // Java List class
class List {
    private Link firstElement = null;

    private class Link { // inner class definition
        public Object value;
        public Link forwardLink;
        public Link backwardLink;

        public Link (Object v, Link f, Link b)
            { value = v; forwardLink = f; backwardLink = b; }

        public void addBefore (Object val)
        {
            Link newLink = new Link(val, this, backwardLink);
            if (backwardLink == null)
                firstElement = newLink;
            else {
                backwardLink.forwardLink = newLink;
                backwardLink = newLink;
            }
        }

        ... // other methods omitted

        }

    public void push_front(Object val)
    {
        if (firstElement == null)
```

```
            firstElement = new Link(val, null, null);
        else
            firstElement.addBefore (val);
    }

    ... // other methods omitted

}
```

Note that the method addBefore references the data field firstElement, in order to handle the special case where an element is being inserted into the front of a list. A direct translation of this code into C++ will produce the following:

```
    // C++ List class
class List {
private:
    class Link;     // forward definition
    Link * firstElement;

    class Link { // nested class definition
    public:
        int value;
        Link * forwardLink;
        Link * backwardLink;

        Link (int v, Link * f, Link * b)
            { value = v; forwardLink = f; backwardLink = b; }

        void addBefore (int val)
        {
            Link * newLink = new Link(val, this, backwardLink);
            if (backwardLink == 0)
                firstElement = newLink; // ERROR !
            else {
                backwardLink->forwardLink = newLink;
                backwardLink = newLink;
            }
        }

        ... // other methods omitted
        };
```

```
public:
    void push_front(int val)
    {
        if (firstElement == 0)
            firstElement = new Link(val, 0, 0);
        else
            firstElement->addBefore (val);
    }
    ... // other methods omitted
};
```

It has been necessary to introduce a forward reference for the Link class, so that the pointer firstElement could be declared before the class was defined. Also C++ uses the value zero for a null element, rather than the pseudo-constant null. Finally links are pointers, rather than values, and so the pointer access operator is necessary. But the feature to note occurs on the line marked as an error. The class Link is not permitted to access the variable firstElement, because the scope for the class is not actually nested in the scope for the surrounding class. In order to access the List object, it would have to be explicitly available through a variable. In this case, the most reasonable solution would probably be to have the List method pass itself as argument, using the pseudo-variable this, to the inner Link method addBefore. (An alternative solution, having each Link maintain a reference to its creating List, is probably too memory intensive).

```
class List {
    Link * firstElement;

    class Link {
        void addBefore (int val, List * theList)
        {
            ...
            if (backwardLink == 0)
                theList->firstElement = newLink;
            ...
        }
    };
public:
    void push_front(int val)
    {
            ...
                // pass self as argument
            firstElement->addBefore (val, this);
```

```
    }
    ... // other methods omitted
};
```

When nested class methods are defined outside the class body, the name may require multiple levels of qualification. The following, for example, would be how the method addBefore would be written in this fashion:

```
void List::Link::addBefore (int val, List * theList)
{
    Link * newLink = new Link(val, this, backwardLink);
    if (backwardLink == 0)
        theList->firstElement = newLink;
    else {
        backwardLink->forwardLink = newLink;
        backwardLink = newLink;
    }
}
```

The name of the function indicates that this is the method addBefore that is part of the class Link, which is in turn defined as part of the class List.

5.8 Static Initialization

WARNING *C++ does not use the message passing syntax for invoking static functions*

Static items in both Java and C++ are elements defined as part of a class description that exist independently of class instances. A static data field, for example, will exist as a single value, regardless of the number of instances that have been created. (Even if no instances have been created). A static member function can be invoked without a receiver. One difference is in the syntax used for the latter operation. In Java the syntax is the same as for message passing, with the class name as the receiver. In C++ a static member function is written as the class name, a pair of colons, and the member function name:

```
d = Math.sqrt (d); // Java - invoke static function sqrt
Date::setDefault(12,7,42); // C++ - use qualified name
```

Another way in which C++ and Java differ is in the way in which static data fields are initialized. In Java they are initialized either as direct assignments in the class body, or in a special static initializer block. The following illustrates both of these examples:

```
class box {
    public box (int v)
```

```
    {
        boxCount++;
        value = v;
        if (v == 0)
            zeroCount++;
    }

    private int value;

        // keep track of number of boxes created
    static private int boxCount = 0;

        // also track number of boxes with value zero
    static private int zeroCount;

    static {
        zeroCount = 0; // initialize the zeroCount field
    }
}
```

In C++ there are also two separate mechanisms that can be used. Data fields that are declared either as const or static and are primitive data types can be defined by an initialization of the data field inside the class, as in Java. This syntax, however, is used only for this one situation. All other initializations are performed outside the class body, using a fully qualified name. The syntax used in the latter is similar to that used with initializations in a declaration; an assignment can be used if there is a single argument, or parenthesis if there are two or more arguments:

```
class box {
public:
    box (int v) : value(v)
        { boxCount++;  if (v == 0) zeroCount++; }

private:
    static int boxCount = 0;
    static int zeroCount;
};

    // global initialization is separate from class
int box::zeroCount = 0;
```

The declaration and use of static methods in C++ is generally similar to the techniques used in Java.

A common use for static data fields in Java is to produce class-specific constants, as in the following example:

```
class coloredBox extends box {
    // define the range of color values
    public static final int Red = 1;
    public static final int Yellow = 2;
    public static final int Blue = 3;

    public coloredBox (int v, int c) { super(v); ourColor = c; }

    private int ourColor;
}
```

The C++ language does not have the keyword final. However, in this case such fields can be written as static integer constants:

```
class coloredBox : public box {
public:
        // define the range of color values
    static const int Red = 1;
    static const int Yellow = 2;
    static const int Blue = 3;

    coloredBox (int v, int c) : box(v), ourColor(c) { }

private:
    int ourColor;
};
```

RULE *Use enumerated data types to describe lists of mutually exclusive alternatives*

An even better solution in this situation is to define a new *enumerated data type*. An enumerated type creates a new data type with a specific set of constant elements. This avoids the inadvertent use of integer constants with arithmetic expressions:

```
class  coloredBox : public box {
public:
    enum Colors {Red, Yellow, Blue};

    coloredBox (int v, Colors c) : box(v), ourColor(c) { }
```

```
private:
    Colors ourColor;
};
```

To access an enumerated constant value that is defined within a class a fully qualified name must be used. An example would be coloredBox::Red.

5.9 Final Classes

C++ does not have the Java concept of a final class, a class that cannot be subclassed. One technique that is sometimes used to achieve this effect is to declare all constructors for the class as private; since subclasses cannot then invoke their parent class constructors, the C++ compiler will not permit them to be written. But this introduces a different problem; namely that such values cannot be created at all. To get around this, the programmer can write *pseudo-constructors*, static functions which do nothing more than invoke a constructor value:

```
class privateBox {
public:
        // pseudo-constructor used for creation
    static privateBox & makeBox(int v) { return privateBox(v); }

private:
        // since constructor is private,
        // cannot create instances directly
    privateBox (int v) : value(v) { }
    int value;
};
```

The static function is then used each time a value is needed:

```
    // make a new box value
privateBox aBox = privateBox::makebox(7);
```

Test Your Understanding

1. What are some of the superficial similarities between a class definition in Java and in C++? What are some superficial differences?

2. What does it mean to say that C++ separates class definition and implementation?

3. What is an inline method? How is it declared?

4. How does the compiler treat the invocation of an inline method differently from the invocation of a normal method?

5. What is the advantage to using inline methods? What is a disadvantage?

6. What is a qualified name?

7. What is a prototype? What information is omitted from a prototype that is found in a function definition?

8. What is indicated by a declaration that uses the extern modifier?

9. Why are forward references necessary in C++ and not in Java?

10. What two tasks are tied together by a constructor?

11. What is a default constructor? What are some of the conditions when it will be invoked?

12. What is a copy constructor? What are some of the conditions when it will be invoked?

13. What types of items can be found in an initializer list?

14. What is the order in which data fields will be initialized?

15. What is a default argument? In what situations can default arguments be used to replace two overloaded function bodies with a single function?

16. What four methods are necessary for a class to be said to satisfy the orthodox canonical class form?

17. What error is being exhibited by the following C++ class definition?

```
class A {
public:
    void a () { ... }

    class B {
    public:
        void b () { a(); }
    };
};
```

18. How is a nested class in C++ different from an inner class in Java?

19. How is the initialization of static data fields in C++ different from that of Java?

20. How can final constants in Java be represented in C++?

21. What is the effect of declaring constructor methods as private?