Chapter 4

Memory Management

In Java the task of memory management is largely conducted in the background, just beyond the issues of concern to the typical programmer. This *can* be an advantage, as it removes a complex task from the programmers range of vision, thus allowing him or her to concentrate on more application specific details. On the other hand, it also means that the Java programmer has little control over how memory management is performed.

Using C++, in contrast, memory management is explicitly under the direction of the programmer. If properly used this can work to the programmers advantage, permitting much more efficient use of memory than is possible with Java. But if improperly used (or, more often, ignored) memory management issues can bloat a program at run-time, or make a program very inefficient. For this reason, to write effective C++ programs it is important to understand how the memory management system operates.

WARNING
The C++ programmer must always be aware of how memory management is being performed

Because in C++ the programmer is responsible for memory management, there are a variety of errors that are possible in C++ programs but are rare or unusual in Java. These include the following:

- Using a value before it has been initialized. (The Java language requires a dataflow analysis that will detect many of these errors. The C++ language makes no such requirement).
- Allocating memory for a value, then not deleting it once it is no longer being used. (This will cause a long running program to consume more and more memory, until it fails in a catastrophic fashion.)
- Using a value after it has been freed. (Once returned to the memory management system, the contents of a freed location are typically overwritten in an unpredictable fashion. Thus, the contents of a freed value will be unpredictable.)

WARNING
C++ does not perform dataflow analysis to detect potentially undefined local variables

Each of these potential errors will be illustrated by examples in the remainder of this chapter.

4.1 The Memory Model

To understand how memory management is performed in C++ it is first necessary to appreciate the C++ memory model, which differs in a fundamental fashion from the Java memory model. In many ways, the C++ model is much closer to the actual machine representation than is the Java memory model. This closeness permits an efficient implementation, at the expense of needing increased diligence on the part of the programmer.

DEFINE *Stack-resident values are created when a procedure is entered, and deleteted when the procedure exits*

In C++ there is a clear distinction between memory values that are *stack-resident*, and those that are *heap-resident*. Stack resident values are those created automatically when a procedure is entered or exited, while heap resident values must be explicitly created using the `new` operator. In the following sections we will describe some of the characteristics of both of these, and the problems the programmer should look out for.

4.2 Stack Resident Memory Values

NOTE *An activation record is sometimes called an activation frame, or a stack frame*

Both the stack and the heap are internal data structures managed by a run-time system. Of the two, the stack is much more orderly. Values on a stack are strongly tied to procedure entry and exit. When a procedure begins execution, a new section of the stack is created. This stack segment holds parameters, the return address for the caller, saved internal registers and other machine specific information, and space for local variables. The section of a stack specifically devoted to one procedure is often termed an *activation record.*

In Java, only primitive values are truly stack resident. Objects and arrays are always allocated on the heap. Compare the following two example procedures. Each creates a local integer, a local array, and an object instance.

```
void test ()    // Java
{
    int i;
    int a[ ] = new int[10];
    anObject ao = new anObject();
    ...
}

void test () // C++
{
    int i;
    int a[10];
    anObject ao;
    ...
}
```

In the Java procedure only the primitive integer value is actually allocated space on the stack by the declaration. Both the array and the object value must be explicitly created (as heap-resident values) using the `new` operator. In C++, on the other hand, all three values will reside on the stack, and none require any actions beyond the declaration to bring them into existence.

Procedure invocations execute in a very orderly fashion. If procedure A invokes procedure B, and B in turn invokes C, then procedure C must terminate before B will continue with execution, and B in turn must terminate before A will resume. This property allows activation records to be managed in a very efficient and orderly manner. When a procedure is called, an activation record is created and pushed on the stack, and when the procedure returns, the activation record can be popped from the stack. A pointer can be used to point to the current top of stack. Allocation of a section of the stack then simply means moving this pointer value forward by the required amount.

The efficiencies of stack memory are not without cost. There are two major drawbacks to the use of the stack for variable storage:

- The *lifetime* of stack resident memory values is tied to procedure entry and exit. This means that stack resident values cease to exist when a procedure returns. An attempt to use a stack resident value after it has been deleted will typically result in error.
- The *size* of stack resident memory values must be known at compile time, which is when the structure of the activation record is laid out.

DEFINE *The lifetime is the period of time a value can be used*

The following sections will describe some of the implications of these two issues.

4.2.1 Lifetime errors

An important fact to remember is that once a procedure returns from execution, any stack resident values are deleted and no longer accessible. A *reference* to such a value (see Chapter 3) will no longer be valid, although it may for a time appear to work. Here are some examples.

```
    // WARNING – Program contains an error
char * readALine ()
{
    char buffer[1000]; // declare a buffer for the line
    gets(buffer); // read the line
    return buffer; // return text of line
}
```

In this example the procedure will return a pointer to the array buffer,[1] however the memory for the buffer will have been deleted as part of the procedure return. The pointer will reference values that may or may not be correct, and will almost certainly be overwritten once the next procedure is called.

Contrast this with an equivalent Java procedure:

```
String readALine (BufferedInput inp) throws IOException
{
    // create a buffer for the line
    String line =  inp.readLine();
    return line;
}
```

Although the variable line is declared inside the procedure, the *value* assigned to the variable will be heap-resident. Therefore this value will continue to exist, even after the procedure returns.

Function return values are by no means the only way to create dangling references. An assignment to a global variable or a by-reference parameter can do the same thing, as shown in the following:

```
char * lineBuffer; // global declaration of pointer to buffer

    // WARNING – Program contains an error
void readALine ()
{
    char buffer[1000]; // declare a buffer for the line
    gets(buffer); // read the line
    lineBuffer = buffer; // set pointer to reference buffer
}
```

Some of the more sophisticated C++ compilers will warn about such errors, but the programmer should not depend upon this being caught.

4.2.2 Size errors – the Slicing Problem

For object-oriented programming, one of the most severe restrictions of stack-based memory management derives from the fact that the positions of values within the activation record are determined at compile time. For primitive values and pointers this is a small concern.

[1]The close relationship in C++ between arrays and pointers is discussed in detail in Chapter 3. It should also be noted that gets is a problematic function, for many of the reasons cited in this chapter. Better solutions to this problem are provided by the Stream I/O package described in Chapter 10. However, gets is part of the legacy C++ inherits from C, and is still commonly found in many programs.

However, it is an issue for arrays and for objects. For arrays, it means that the array bound must be known at compile time. For objects, it means a limitation on the degree to which values can be polymorphic.

Array Allocations

Stack resident arrays must have a size that is known at compile time. Often programmers avoid this problem by allocating an array with a size that is purposely too large. An earlier example, which we will repeat, illustrated this technique. Here the programmer is reading lines of input from a file, and has allocated the character array to hold 200 elements, on the assumption that no line will have any more than this number of characters.

NOTE *The size of any value stored on the stack must be known at compile time*

```
    // WARNING – Program contains an error
char buffer[200]; // making array global avoids deletion error
char * readALine ()
{
    gets(buffer); // read the line
    return buffer;
}
```

Potential problems occur since array bounds are not checked in C++ at run time[2]. Should an overly-long line be encountered in the file, the buffer will simply be exceeded, and the values read will flow over into whatever happens to follow the array in the activation record. This will have the effect of changing the values of other variables in an unpredictable fashion, with generally infelicitous results.

RULE *Never assume that just because an array is big, it will always be "big enough"*

It *is* possible to create arrays with sizes determined at run-time using heap-resident values. This is done in a fashion similar to Java, with the exception that the programmer must explicitly free the memory when no longer required (see Section 4.3).

```
char * buffer;
int newSize;
...      // newSize is given some value
buffer = new Char[newSize]; // create an array of the given size
...
delete [ ] buffer; // delete buffer when no longer being used
```

```
class A {
public:
        // constructor
    A () : dataOne(2) { }

        // identification method
    virtual void whoAmI () { printf("class A"); }
private:
    int dataOne;
};

class B : public A {
public:
        // constructor
    B () : dataTwo(4) { }

        // identification method
    virtual void whoAmI () { printf("class B"); }
private:
    int dataTwo;
};
```

Figure 4.1: A Class Hierarchy for Illustrating the Slicing Problem

The Slicing Problem

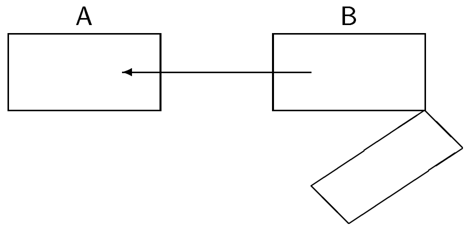
Java programmers are used to object values being *polymorphic* (see Chapter 6). That is, a variable declared as maintaining a value of one class can, in fact, be holding a value derived from a child class. In the class hierarchy shown in Figure 4.1,[3] class A is extended by a new class B, which overrides the virtual method whoAmI.[4] In both Java and C++ it is legal to assign a value derived from class B to a variable declared as holding an instance of class A:

```
A instanceOfA;    // declare instances of
B instanceOfB;    // class A and B

instanceOfA = instanceOfB;

instaceOfA.whoAmI(); // question: what will happen?
```

However, the effect of this assignment will differ in the two languages. The Java programmer will expect that, while the declaration of instanceOfA is class A, the value will continue to be that of a B, and thus the method invoked will be that found in class B.

This polymorphic behavior runs into conflict with the stack based memory allocation model. Note that values of class B are larger than values of class A, since they include an additional data field (namely, the inherited field dataOne, and the field dataTwo defined in class B). In order to maintain the efficiencies of the stack-based memory allocation, these additional fields are simply sliced away when the assignment occurs. Thus, the value ceases to be an instance of class B, and simply becomes an instance of class A:

RULE
Static variables are never polymorphic



In this light, it is therefore not surprising that the method executed by the invocation of whoAmI will be that of class A, and not that of class B. This is true regardless of whether the method whoAmI was declared virtual.

[2]See Chapter 3 for a further discussion on this issue. Note that there is an alternative function, fgets, that permits the buffer size to be specified. As a general matter, fgets should be preferred over gets for just this reason. However, the type of error discussed here arizes in many situations, so the point is still valid.

[3]Output is here produced using the statement printf, which is one of two output techniques commonly encountered in C++ programs. The topic if I/O libraries will be discussed in Chapter 10.

[4]See Chapter 6 for a more complete discussion of the keyword virtual.

WARNING *The message passing rules for pointers and referenes are different from those for simple variables*

Note carefully, however, that slicing does not occur with references or with pointers:

```
A & referenceToA = instanceOfB;
referenceToA.whoAmI(); // will print class B

B * pointerToB  = new B();
A * pointerToA = pointerToB();
pointerToA->whoAmI(); // will print class B
```

Slicing only occurs with objects that are stack resident. For this reason, many C++ programs make the majority of their objects heap resident. The issues the Java programmer learning C++ should be aware of when using heap allocation are discussed in detail in the next section.

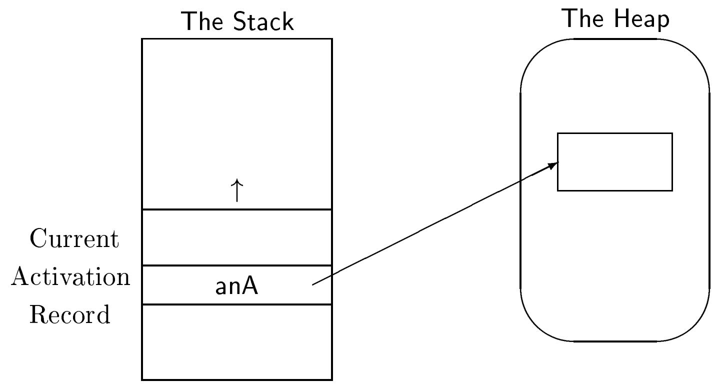
4.3 Heap Resident Memory Values

Heap resident values are created using the new operator. Memory for such values resides on the *heap*, or *free store*, which is a separate part of memory from the stack. Typically such values are accessed through a pointer, which will often reside on the stack. The following contrasting code fragments in C++ and Java will illustrate this:

NOTE *The parenthesis are omitted from a* new *statement if there are no arguments for the constructor*

```
void test () // Java
{
    A anA = new A();
    ...
}

void test () // C++
{
    A * anA = new A; // note pointer declaration
    if (anA == 0) ... // handle no memory situation
    ...
    delete anA;
}
```

Here anA will reside on the stack (in both Java and C++) but the value it references will reside on the heap:



The Java language hides the use of this pointer value, and programmers seldom need be concerned with it. In C++, on the other hand, the pointer declaration is explicitly stated.

In Java if a memory request cannot be satisfied an OutOfMemmory exception is thrown. Newer versions of C++ will likewise throw a bad_alloc exception. However, earlier versions of the language would simply return a null pointer value when a request could not be satisfied. It is therefore important to test for this condition, and take appropriate action (such as throwing an exception yourself) if it occurs.

WARNING *Not all compilers will throw an exception when memory becomes exhausted*

A more important difference relates to the recovery of heap based memory. Java incorporates *garbage collection* into its run-time library. The garbage collection system monitors the use of dynamically allocated variables, and will automatically recover and reuse memory that is no longer being accessed. C++, on the other hand, leaves this task to the programmer. Dynamically allocated memory must be handed back to the heap manager using the delete operator. There are two forms of this operator, both of which have been used in examples presented earlier in this chapter. The deletion of an individual object is performed by simply naming the pointer variable, as in the example shown above. When deleting an array a pair of empty square braces must be used. An example showing this syntax was presented earlier in Section 4.2.2.

DEFINE *A garbage collection system searches out and recovers unused memory*

Because memory recovery must be an explicit concern of the programmer, four types of errors are common:

- Forgetting to allocate a heap-resident value, and using a pointer as if it were referencing a legitimate value.
- Forgetting to hand unused memory back to the heap manager.
- Attempting to use memory values after they have been handed back to the heap manager.
- Invoking the delete statement on the same value more than once, thereby passing the same memory value back to the heap manager.

In Java the first type of error, if not caught by the compiler, will generally raise a null pointer exception. C++ compilers are not obligated to try and detect the use of variables before they have been set, and few will. Furthermore, the initial contents of memory are generally not determined. Thus, an uninitialized pointer value will sometimes contain a legal memory address even before it has been set, although there is no way to know where in memory it is pointing to. Attempting to read from such a value or assigning to it will cause an unpredictable result.

WARNING *Uninitialized values in C++ are not only not reported by the compiler, but their initial values are generally garbage*

DEFINE *A memory leak is an allocation of memory that is never recovered*

The second type of error is termed a *memory leak*. Often such leaks can occur without any harmful effects. However, in a long running program or if memory allocation occurs in a situation that is executed repeatedly, such leaks will cause the memory requirements for the program to increase over time. Eventually the heap manager will be unable to service a request for further memory, and the program will halt. Leaks are often the result of successive assignments to the same pointer variable:

```
AnObject * a;
...
a = new AnObject();
...
a = new AnObject(); // leak, old reference is now lost
```

The third type of error sometimes occurs as the result of an over-zealous attempt to avoid the second error. Here memory is passed back to the heap manger before all references to the value have been deleted. As part of managing and recycling heap resident values, the heap manager often stores pertinent information in the value. For example, the heap manager may keep a list of similarly sized blocks of memory, and store in each block a pointer to the next element. Thus, after a value is deleted the contents are often overwritten. Reading from such a value will produce garbage, and writing to such a value will confound the heap manager. Both errors are typically catastrophic.

Depending upon the sophistication of the memory manager, the fourth error may or may not be severe. Some heap mangers can detect this condition. Other heap mangers will not notice the error, but the internal data structures used by the heap manager will be put into an inconsistent state. This, too, can result in unpredicatable errors.

RULE *Always match memory allocations and deletions*

A simple rule of thumb is that every time the new operator is used, the programmer should be able to identify where and under what circumstances the associated delete directive will be issued. There are two techniques that are frequently used to help simplify the management of heap values. These are:

- Hide the allocation and release of dynamic memory values inside an object. The object is therefore the “owner” of the heap resident value, and is “responsible” for memory management. For example, the memory management is often tied to the lifetime of the object, which can sometimes be more reliably predicted (for example, the object is stack resident). This technique is only applicable where there is only one pointer to the heap resident value.

- If it is not possible to designate a single "owner" for a heap resident value, then it is difficult to know who should be responsible for deleting the value once it is no longer being referenced. This problem can be solved by maintaining, as part of the heap, a *reference count* that indicates the number of pointers to the value. When this count is decremented to zero, the value can be recovered.

We will illustrate each of these techniques by developing a data abstraction for the *String* data type.

4.3.1 Encapsulating Memory Management

String literals in C++, unlike Java, are very low level abstractions. A string literal in C++ is treated as an array of character values, and the only permitted operations are those common to all arrays. The String data type in Java is designed to provide a higher level of abstraction. A version of this data structure is provided in the new Standard Template Library (the STL, see Chapter 9). It is also a common example found in many introductory C++ texts. The version we present here is skeletal, describing only those features related to memory management. More complete implementations can be found described in [Budd 98a, Budd 94, Lippman 91, Coplien 92].

The class description for String is shown in Figure 4.2. As is common in C++, the shorter methods (in this case, all but one method) are provided using in-line definitions in the class heading, while longer methods are found in an implementation file, outside the class definition.

String variables can be declared with no initialization, or initialized using either a literal string text, a character array (that is, a pointer to a character) or another string value:

```
string a;
string b = "abc";
string c = b;
```

The important feature of this string abstraction is that there is a one-to-one matching of String objects to dynamically allocated buffers. Every string has one buffer, and every buffer is "owned" by a specific String. This buffer is created by the method resize:

NOTE *One way to manage dynamically allocated memory is to make an object responsible for managing the memory*

```
void String::resize(int size)
{
    if (buffer == 0) // no previous allocation
        buffer = new char[1 + size];
    else if (size > strlen(buffer)) {
```

```
class String {
public:
        // constructors
    String    () : buffer(0) { }
    String    (const char * right) : buffer(0)
        { resize(strlen(right)); strcpy(buffer, right); }
    String    (const String & right) : buffer(0)
        { resize(strlen(right.buffer)); strcpy(buffer, right.buffer); }

        // destructor
    ~String () { delete [ ] buffer; }

        // assignment
    void    operator = (const String & right)
        { resize(strlen(right.buffer)); strcpy(buffer, right.buffer); }

private:
    void    resize (int size);
    char *     buffer;

};
```

Figure 4.2: The Class String (Version One)

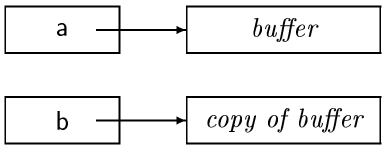
```
            delete [ ] buffer; // recover old value
            buffer = new char[1 + size];
        }
    }
```

If no buffer has yet been allocated a buffer of the requested size is created. (The one extra character is for the null character inserted at the end of a string by the function strcpy. Otherwise, if the current buffer is too small for the requested number of characters, then the current buffer is returned to the heap manager, and a new buffer is created.

A string variable can be assigned a value from another string variable:

```
a = b;
```

In this case, a *copy* of the contents of the right hand side is created:



We have seen how the dynamic buffer allocation will take place in the constructor. We have seen how this buffer may be deleted and replaced with another as the result of an assignment. If we now ask our fundamental question, for every new is there a matching delete, we see that there is one remaining case we have not yet discussed. What happens if a variable of type String is destroyed? This will happen, for example, when local variables are recovered at the end of a procedure. To handle this case, C++ provides a mechanism to perform actions immediately before the point of destruction. This ability is provided by means of a procedure called the *destructor*. The destructor for a class is a non-argument procedure with a name formed by prepending a tilde before the class name. The destructor shown in Figure 4.2 simply deletes the dynamically allocated buffer. With this facility, we can now match all allocations and deletes, ensuring no memory leaks will occur.

DEFINE *A destructor is a procedure that performs whatever housekeeping is necessary before a variable is deleted*

Do not confuse the destructor with the delete operator. The delete operator is the function used to actually return memory to the heap manger. The destructor is charged with whatever "housecleaning" tasks are necessary before a variable disappears. Often this housecleaning involves memory management, but not always. In the next section we will encounter a destructor that performs more than simply memory management.

The concept of a destructor in C++ should also not be confused with the notion of a finalize method in Java. Destructors are tied to *variables*, while the finalize method is tied to a *value*. The destructor will be invoked when a variable is about to be destroyed, for example as soon as it goes out of scope when a procedure returns, or when the variable itself is dynamically allocated and has been the target of a delete. A finalize method in

Java is invoked when the value holding the method is about to be recovered by the garbage collector. There is no guarantee in Java when this will occur, if ever. Thus programmers cannot make assumptions concerning behavior in Java based on the execution of code in a finalize method. In this regard the C++ programmer is on slightly safer ground, as the rules for when a destructor will be invoked are carefully spelled out by the language definition.

An Execution Tracer

NOTE *Constructors and destructors can be used for a variety of nested activities*

A clever example that illustrates the use of constructors and destructors is an execution tracer. The class Trace takes as argument a string value. The constructor prints a message using the string, and the destructor prints a different message using the same string:

```
class Trace {
public:
        // constructor and destructor
    Trace (string);
    ~Trace ();
private:
    string name;
};

Trace::Trace (string t) : name(t)
{
    cout << "Entering " << name << endl;
}

Trace::~Trace ()
{
    cout << "Exiting " << name << endl;
}
```

To trace the flow of function invocations, the programmer simply creates a declaration for a dummy variable of type Trace in each procedure to be traced:

```
void procedureOne ()
{
    Trace dummy("Procedure One");
    ...
    procedureTwo(); // proc one invokes proc two
}

void procedureTwo ()
```

```
{
    Trace dummy("Procedure Two");
    ...
    if (x < 5) {
        Trace dumTrue("x test is true");
        ...
        }
    else {
        Trace dumFalse("x test is false");
        ...
        }
    ...
}
```

Using the ability to declare variables local to a block, trace variables can even be used to follow the effect of conditional or loop statements. by their output, the values of type Trace will trace out the flow of execution. A typical output from this program might be:

```
Entering Procedure One
Entering Procedure Two
Entering x test is true
...
Exiting x test is true
Exiting Procedure Two
Exiting Procedure One
```

The auto_ptr Class

The relationship between a string value and the underlying buffer is a pattern that is repeated many times in programs. That is, there is an object that must dynamically allocate another memory value in order to perform its intended task. However, the lifetime of the dynamic value is tied to the lifetime of the original object; it exists as long as the original objects exists, and should be eliminated when the original object ceases to exist.

To simplify the management of memory in this case, the standard library implements a useful type named auto_ptr (see Section A.5 in Appendix A). A simplified version of auto_ptr could be described as follows:[5]

```
template <class T>
class auto_ptr {
```

[5]This class description uses templates, which will be described in Chapter 9.

```
public:
    auto_ptr (T *p) : ptr(p) { }
    auto_ptr () : ptr(0) { }
    ~auto_ptr() { delete ptr; }

    void operator = (T* right)
    {
        delete ptr;
        ptr = right;
    }
private:
    T * ptr;
};
```

The actual class is more robust and will handle assignments and copies, but this form illustrates the key features. An auto ptr object simply holds a pointer value, and will delete the memory referenced by the pointer when it itself is destroyed.

A revised string class that used auto pointers would look similar to the following:

```
class string {
    ...
private:
    auto_ptr<char> buffer;
};

void String::resize(int size)
{
    if ((buffer == 0) || (size > strlen(buffer)))
        buffer = new char[1 + size];
}
```

In this form it would not be necessary to implement a destructor for the string class, since the default destructor would invoke the destructor for the auto pointer field, which would in turn return the buffer memory back to the heap manager.

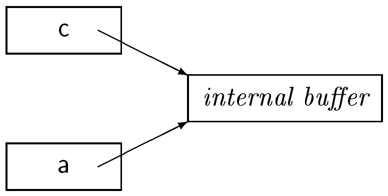
Auto pointers should be used any time there is a one-to-one correspondence between objects and an internal heap-allocated memory, and the lifetime of the internal object is tied to the lifetime of the surrounding container.

4.3.2 Reference Counts

There are many situations where two or more objects need to share a common data area. We could imagine, for example, wanting to change the semantics for the assignment of strings so that two strings would share a common internal buffer. That is, subsequent to the statement:

```
a = c;
```

both variables a and c would reference the same buffer:



The difficulty with this interpretation is that we no longer have a single unambiguous object that can be said to "own" the dynamically allocated value, and can therefore be charged with disposing of it when it is no longer needed. A solution to this problem is to augment the dynamic value with a count of the number of pointers that reference it. This count is termed a *reference count*. Care is needed to ensure the count is accurate; whenever a new pointer is added the count is incremented, and whenever a pointer is removed the count is decremented.

DEFINE *A reference count is the count of the number of pointers to a dynamically allocated object*

Figure 4.3 shows a revision of the String abstraction that incorporates these changes. The method resize has here been replaced with reassign. The method reassign replaces the current string reference with another. In doing so, it both decrements the reference count on the old string reference, and increments the count on the new. Performing the increment first ensures the procedure will work in the special case where a variable is assigned to itself. If the reference count for the old value becomes zero, the memory for the entire string reference is returned to the heap manager, using the method delete. Before recovering the memory, the heap manger will execute the destructor for the string reference being deleted. This destructor will, in turn, return the buffer to the heap manager.

```
void String::reassign(String::StringReference * np)
{
    if (np) // increment count on new value
        np->count += 1;
    if (p) { // decrement reference counts on old value
        p->count -= 1;
        if (p->count == 0)
            delete p;
    }
    p = np; // change binding
}
```

```
class String {
public:
        // constructors
    String () : p(0) { }
    String (const char * right) : p(0)
        { reassign(new StringReference(right)); }
    String (const String & right) : p(0) { reassign(right.p); }

        // destructor
    ~String    () { reassign(0); }

        // assignment
    void operator = (const String & right) { reassign(right.p); }

private:
    void reassign (StringReference *);

    class StringReference {
        public:
            int count;
            char * buffer;
            StringReference(const char * right);
            ~StringReference() { delete [ ] buffer; }
    }

    StringReference * p;
}

void String::StringReference::StringReference(const char * right)
{
    count = 0;
    buffer = new Char[1 + strlen(right)];
    strcpy(buffer, right);
}
```

Figure 4.3: The Class String (Version Two)

The structure of the object holding both the reference count and the buffer is defined by means of a nested class declared within the body of the String class. The reader should note that although there are superficial syntactic similarities, there are some semantic differences between nested classes in C++ and inner classes in Java. These are discussed in more detail in Chapter 5.

The values held by reference counts can be illustrated by tracing the execution of a simple program. Imagine the following:

```
string g; // global string value

void test ()
{
    string a = "abc";
    string b = "xyz";
    string c;
    c = a;
    a = b;
    g = b;
}
```

The following table summarizes the reference counts associated with the values ”abc” and ”xyz” as the program executes:

statement	"abc"	"xyz"
`string a = "abc";`	1	0
`string b = "xyz";`	1	1
`string c;`	1	1
`c = a;`	2	1
`a = b;`	1	2
`g = b;`	2	2
end of execution:		
destructor for `c`	1	2
destructor for `b`	1	1
destructor for `a`	1	0

We are left with a single reference (namely the global variable `g`) pointing to the value `"abc"`. The value `"xyz"` will have been recovered when the reference count reached zero, that is, when the variable `a` was deleted.

Test Your Understanding

1. What are some of the advantages of putting the programmer in control of memory management? What are some of the disadvantages?

2. What is a stack resident value? When is it allocated? When is it deleted?

3. What are characteristics of stack resident values that are not necessarily characteristics of heap resident values?

4. Explain the error in the following program fragment:

```
char * secretMessage()
{
    char messageBuffer[100];
    strcpy (messageBuffer "Eat Ovaltean!");
    return messageBuffer;
}
```

5. Explain the slicing problem, and under what circumstances it will occur.

6. What is a heap resident memory value? How is this value allocated?

7. What are the four types of errors that can occur in the recovery of heap resident values?

8. What is a destructor? When is it invoked?

9. What is a reference count?

10. The following statement is legal. It will create a temporary string value for the right hand expression, then assign the temporary to the left hand variable, before destroying the temporary. Assuming the reference counting scheme is being used to implement the string data type, trace the reference counts on the various internal buffer values.

```
string text = "initial text";
text = "new text";
```