How do we delay **delete** to the commit function? What do we put in the forward and reverse functions?

First, consider that in C++ **new** does two things – allocation and construction. Similarly, **delete** performs deallocation and destruction.

An example of why this is an issue:

struct A

{

static int activeCount;

A() { activeCount++; }

~A()

{

--activeCount;

if (activeCount == 0)

{

//Change the simulation state here

}

}

};

void eventMethod(A\* pa)

{

...

if (...)  
 delete pa;

...

}

We want to delay the *deallocation* of pa until the commit method, but we do want to apply the side effects of its destruction immediately.

What first comes to mind is calling the destructor explictly in the forward event, without deallocating the memory:

void eventMethod\_forward1(A\* pa)

{

...

if (...)

{

pa->~A();

}

}

Then, in the commit method we must perform the actual memory deallocation. However, if we call **delete pa**, the destructor will be called again (remember, we called it in the forward event.) Calling the destructor again is definitely an error, because it would apply its side effects twice.

Another option we might try is calling **free(pa)** in the commit method. However, calling **free** on a pointer allocated with C++’s standard **new** operator is NOT permitted.

Solution 1

Install a global memory allocator that replaces operator **new** in all cases. This is even if there are custom such operators defined.

void\* operator new(size\_t size)

{

return malloc(size);

}

Now, since all memory is allocated with **malloc()**, we can call **free()** in the commit function. We are able to deallocate the object without invoking its destructor. Also, there is the added benefit that the commit function need not know the type of pointer beeing freed -> we just call **free()** on every pointer in the stack.

Problem with solution 1

Sometimes, the semantics of the C++ program may depend on where its objects reside. It’s common practice for C++ projects to implement a custom allocator that uses a memory pool. For example, ROSE uses this approach; all AST nodes reside in a centralized memory pool. The code then traverses the memory pools as a means of finding all instances of a certain object. Changing the allocator would break ROSE in this case.

Solution 2

Use [placement new](http://www.parashift.com/c++-faq-lite/dtors.html#faq-11.10). We explicitly call **malloc** at each allocation site, and then invoke placement new. When we encounter **delete**, again we call the destructor instead. In commit, we call free.

Problem with solution 2

Same problem as solution 1. We mess up custom allocation pools.

Solution 2.1

For each allocation site, decide if the default **new** operator is being called or a custom operator is called. If it’s the default **new**, keep use solution 2 (or 1). If there is a custom **new**, say **new\_custom**, we invoke **new\_custom**.

If there is a custom **new**, there must also be a custom **delete**, say **delete\_custom**. The signature of custom delete is **void (void\* p)**. In other words, we can call **delete\_custom** in the commit method instead of calling **free**, while still avoiding the extra invocation of A’s destructor from the commit method. A’s destructor is called explicitly in the forward method, as in solutions 1 & 2.

Problem with solution 2.1

Does the C++ syntax allow invoking the **delete\_custom** operator in a way that does not trigger the invocation of the destructors?

Yes it does! Call **operator new()** and **operator delete()** instead of using the **new** or **delete** keywords.

Solution 3

This solution unifies solutions 1 & 2 without requiring special treatment of custom allocators. It also preserves the semantics of custom allocators and does not require modifying all allocation sites.

Recall our example event which has the line

**delete pa;**

In the forward event, we call the destructor explicitly

**pa->~A();**

In the commit event, we call the deallocator explicitly (nice feature of C++ I didn’t know about)

**operator delete(pa)**

Another nice aspect of this solution is that the commit function need not know the types of the pointers it must deallocat, because **operator delete** has a formal parameter of **void \***

Calls to **delete** and the reverse function

We’ve established that all calls to **delete pa** in the forward method will be replaced with explicit calls to the destructor, **pa->~A()**. This makes the destructor a function call that must be reversed in the reverse method.

So, in the reverse method, for each **delete** in the original event method, we call the generated *reverse destructor*.

Generating a reverse destructor can be quite tricky, however. When **pa->~A()** is invoked, the compiler automatically calls destructors for all the superclasses of A, as well as destructors for all members of A. Destructors for members of A may recursively call more destructors for their superclasses and members. When we generate the reverse destructor, we must mimic the compiler’s recursive invocation of destructors.