Document Numbers: X3J16/95-0042
WG21/N0642
Date: January 31, 1995
Reply To: Bill Gibbons
bgibbons@taligent.com

Placement Delete

Introduction

When placement new is used to allocate space, there is the potential for exceptions to cause memory leaks. There are several possible solutions. One was explored in 94-0104/N0491, but was rejected by the extensions WG. Another solution, placement delete, was found more acceptable.

The Problem

A *new-expression* for a class type implicitly calls operator new to allocate memory and then implicitly calls a constructor. If the constructor exits via an uncaught exception, there is no way for the program to explicitly deallocate the now unused memory.

For non-placement *new-expressions*, this problem was solved by requiring the implementation to implicitly call operator delete when propagating such an exception.

This is a simple and reasonably complete solution for non-placement new. But it doesn't work for placement new, because placement new may not actually allocate memory. (The working paper states that the result of calling operator delete on a pointer returned from placement new is undefined.) And so operator delete is not called automatically during exception handling for a placement-style new-expression .

```
class PrivateHeap;

struct A {
      A() { throw “Boom!”; }
      operator new(size_t, PrivateHeap *);
};

void f(PrivateHeap *heap) {
      A *a = new (heap) A;
```

This code leaks memory each time function “f” is called. This is problem we're trying to solve.

The Solution

There is no way for the implementation to know whether a given placement operator new function actually allocates space. Therefore there is no way to know whether it is appropriate to pass the result of placement new to an ordinary operator delete.

The solution is to provide a different operator delete - a “placement delete” - which is a match for the placement new. The placement delete may or may not deallocate memory; that decision is now totally under the control of the programmer. But the implementation can safely assume that it’s safe to call placement delete after placement new.

Placement operator new functions are distinguished by the presence of additional parameters (beyond the allocation size). The obvious way to write a placement operator delete is to include the same additional parameters as in the corresponding placement operator new. These values may be needed by the delete anyway; for example, they may specify a private heap from which the memory is to be allocated (and to which it must be returned).

For example:

```
class PrivateHeap;

struct A {
      A() { throw “Boom!”; }
      operator new(size_t, PrivateHeap *);
      operator delete(void *, PrivateHeap *);
};

void f(PrivateHeap *heap) {
      A *a = new (heap) A;
}
```

Since the placement operator delete function is an exact match for the placement operator new function, the implementation must call the delete function when propagating an exception. This is exactly the same semantics as non-placement new.

Placement operator delete is never called implicitly except in the above situation. In particular, there is no placement syntax for *delete-expression.* There are no syntax changes to the language at all.

Name Lookup

It seems reasonable to require that the matching operator delete have exactly the same extra parameters as the operator new. But must it be in exactly the same scope?

This problem is very similar to the one for ordinary class-scope operator new and operator delete. In that case, there is no requirement that operator new and operator delete be found in the same class, or even that they both be members.

The parallel for placement delete would have the name lookup done using the ordinary lookup rules, and then any exactly matching placement delete in that scope would be used.

The disadvantage is that for a class with a placement opertor new but no placement operator delete, it’s necessary to inspect the base classes and global scope to determine whether placement delete will be used.

In this proposal the former solution (general lookup) is used, but this point is not crucial to the proposal.

Matching New and Delete

Given a placement *new-expression*, and one or more declarations from the lookup of "`operator delete`", exactly how is the appropriate placement operator delete to be chosen for use in exception handling?

The two obvious choices are:

- Using the original arguments, perform overload resolution again.
- Only use a placement `operator delete` with exactly the same additional parameters as in the placement `operator new` chosen for use in this expression.

For this proposal, we've taken the second approach: exact match of additional parameters.

Again, this point is not crucial to the proposal.

Conflicting Signature

Unfortunately, there is already a version of operator delete which has an additional parameter. As described in 3.6.3.2, when `operator delete` is declared as a member function it may have an additional parameter of type `size_t`.

So the following case has a potential ambiguity:

```
struct T {
      void * operator new(size_t, size_t);
      void   operator delete(void *, size_t);
};
```

Is the `operator delete` a placement delete or a plain delete with an extra parameter? There are a several possible ways to resolve this:

- It's a non-placement `operator delete`, unless there is another plain `operator delete` (without the extra parameter) also declared in the class; in which case, it's a placement `operator delete`. This is compatible with existing code, since you can't declare both functions in the same class today.
- It's a non-placement `operator delete`, unless there is an exactly matching placement `operator new` in the class. This breaks existing code because it changes the current meaning of the above example.
- It's a non-placement `operator delete`, because the extra parameter is required for placement delete (for just class members or all versions).

The first solution is proposed, since it is compatible with existing code and simple enough.

Parameter Passing

When placement `operator delete` is called, the same arguments which were passed to placement operator new are passed to `operator delete`. Is there any problem with this?

No, not really. The original arguments which were used to initialize the parameters of `operator new` still exist. They can just be used again to initialize the parameters of `operator delete`. The only issue is whether additional copying is permitted.

Since there are already rules for whether an implementation may copy an argument while initialzing a parameter, we can just extend those rules, and say that if the implementation is allowed to make a copy of one of the `operator new` arguments as part of the `operator new` call, it is allowed to make a copy (of the same original value) as part of the `operator delete` call. If the copy is elided in one place, it need not be elided in the other.

Explicit Calls to Placement Delete

Although there is no proposal for extending the syntax of *delete-expression* to invoke placement operator delete, there is no reason why explicit calls should not be allowed:

```
struct T {
      void operator delete(void *, double);
};
void g(void *p) {
      T::operator delete(p, 1.0);
}
```

Making the above example ill-formed would require special rules and would serve little purpose. This proposal would allow the above example.

Placement Array Delete

All of the changes proposed for non-array placement new/delete would also apply to array placement new/delete. There are no issues specific to array allocation.

The Proposal

We propose the following changes to the working paper, with the understanding that additional changes needed for consistency with this paper and with the working paper are at the editor’s discretion.

<u>3.6.3.2/2</u>

Change: a second parameter of type `size_t` may be added but deallocation functions may not be overloaded.

to: a second parameter of type `size_t` may be added. If both versions are declared in the same class, the one-parameter form is the usual deallocation function and the two-parameter form is used for placement delete [expr.new]. If the second version is declared but not the first, it is the usual deallocation function, not placement delete.

<u>5.3.4/18</u>

add: If the *new-expression* contains a *new-placement*, and a placement operator delete is visible from the appropriate scope and it matches the placement operator new used in this *new-expression*, the placement operator delete is called. [basic.atc.dynamic.deallocation]

A placement operator delete is a match for a placement operator new when it has the same number of parameters and all parameter types except the first are identical (except for top-level qualifiers).

If placement operator delete is called, it is passed the same arguments as are passed to placement operator new. If the implementation is allowed to make a copy of one of the placement new parameters as part of the placement new call, it is allowed to make a copy (of the same original value) as part of the placement delete call, or to reuse the copy made as part of the placement new call. If the copy is elided in one place, it need not be elided in the other.