J16/97-0089 = WG21/N1127
September 3, 1997
Dan Saks & William M. Miller

Relaxing Restrictions on Operators `new` and `delete`

This document is a revision of J16/97-0056R1 = WG21/N1094, which was distributed in the post-London mailing. It differs from that document only in the addition of a proposed modification in 3.7.3 [basic.stc.dynamic], addressing the concern that was raised in London regarding the linkage conflict between a static allocation or deallocation function and its automatically-predeclared counterpart.

In comment Ireland 1, the Irish National body requested the removal of the restriction against operators `new` and `delete` being declared `static` or `inline`. Although there was no explicit requirement in the comment that these operators be allowed in namespace scope, it seemed more consistent to relax that restriction as well. The following proposal implements these changes.

In 3.7.3 [basic.stc.dynamic], replace the first half of paragraph 2 (through the list of implicitly-declared functions) with the following:

> The library provides default definitions for the global allocation and deallocation functions. Some global allocation and deallocation functions are replaceable (18.4.1). Replacements of these functions are subject to the requirements of the One Definition Rule (3.2); a global definition of one of these functions replaces the corresponding version provided in the library (17.3.3.4).
>
> The following allocation and deallocation functions (18.4) are implicitly declared in global scope in each translation unit of a program:
>
> ```
> void* operator new(std::size_t) throw(std::bad_alloc);
> void* operator new[](std::size_t) throw(std::bad_alloc);
> void operator delete(void*) throw();
> void operator delete[](void*) throw();
> ```
>
> That is, a reference to one of these functions from a point at which no corresponding declaration is visible is treated as if the function were declared in global scope with external linkage. [*Example:*
>
> ```
> // no declaration of operator new(std::size_t)
> void f() {
>     int* p = new int;  // uses global operator new
> }
>
> // Error: already (implicitly) declared with external linkage:
> static void* operator new(std::size_t) throw(std::bad_alloc) {
>     // ...
> }
>
> // Internal linkage fine, no preceding use or declaration:
> static void* operator new[](std::size_t) throw(std::bad_alloc) {
>     // ...
> }
>
> void g() {
>     int* q = new int[10];     // uses static operator new[]
> }
> ```
>
> —*end example*]

In 3.7.3.1 [basic.stc.dynamic.allocation], change the first sentence in paragraph 1 from:

> An allocation function shall be a class member function or a global function; a program is ill-formed if an allocation function is declared in a namespace scope other than global scope or declared static in global scope.

to:

> An allocation function may be a class member or a function at namespace scope; a program is ill-formed if an allocation function is declared inline and not static in global namespace scope.

In 3.7.3.2 [basic.stc.dynamic.deallocation], change the first sentence in paragraph 1 from:

> Deallocation functions shall be class member functions or global functions; a program is ill-formed if deallocation functions are declared in namespace scope other than global scope or declared static in global scope.

to:

> A deallocation function may be a class member function or a function at namespace scope; a program is ill-formed if a deallocation function is declared inline and not static in global namespace scope.

In 5.3.4 [expr.new], add following the first sentence of paragraph 1:

> The type of that object is the *allocated type*.

In the same section, replace paragraphs 9 through 11 with the following:

> A *new-expression* obtains storage for the object by calling an allocation function (3.7.3.1). If the *new-expression* terminates by throwing an exception, it may release storage by calling a deallocation function (3.7.3.2). If the allocated type is a non-array type, the allocation function's name is `operator new` and the deallocation function's name is `operator delete`. If the allocated type is an array type, the allocation function's name is `operator new[]` and the deallocation function's name is `operator delete[]`.
>
> A *new-expression* passes the amount of space requested to the allocation function as the first argument of type `std::size_t`. That argument shall be no less than the size of the object being created and may be greater than the size of the object being created only if the object is an array. If present, the *new-placement* supplies additional arguments to the allocation function call.
>
> An implementation shall provide default definitions for the global allocation functions (3.7.3, 18.4.1.1, 18.4.1.2). [Note: a C++ program can provide alternative definitions of these functions (17.3.3.4), and/or class-specific versions (12.5).]

In the same section, add the following after paragraph 12:

If the *new-expression* begins with a unary `::` operator, the allocation function's name is looked up in the global scope. Otherwise, if the allocated type is a class type `T` or an array thereof, the allocation function's name is looked up in the scope of T. If this lookup fails to find the name, or if the allocated type is not a class type, the allocation function's name is looked up in the context of the *new-expression* following the normal rules for name lookup (3.4.1), ignoring member functions that might be found in lexically-enclosing class scopes. [Note: Argument-dependent name lookup (3.4.2) is not performed.] Overload resolution (13.3) selects the appropriate allocation function.

In the same section, replace paragraphs 17 through 19 with the following:

If any part of the object initialization described above *[Footnote: This may include evaluating a new-initialization and/or calling a constructor.]* terminates by throwing an exception and a suitable deallocation function can be found, the deallocation function is called to free the memory in which the object was being constructed, after which the exception continues to propagate in the context of the *new-expression*. If no unambiguous matching deallocation function can be found, propagating the exception does not cause the object's memory to be freed. [Note: this is appropriate when the called allocation function does not allocate memory; otherwise, it is likely to result in a memory leak.]

If a *new-expression* calls a deallocation function, it passes the value returned from the allocation function call as the first argument of type `void*`. If present, the *new-placement* supplies additional arguments to the deallocation function call.

If the *new-expression* begins with a unary `::` operator, the deallocation function's name is looked up in the global scope. Otherwise, if the allocated type is a class type `T` or an array thereof, the deallocation function's name is looked up in the scope of `T`. If this lookup fails to find the name, or if the allocated type is not a class type or array thereof, the deallocation function's name is looked up in the context of the *new-expression* following the normal rules for name lookup (3.4.1), ignoring member functions that might be found in lexically-enclosing class scopes. Overload resolution (13.3) is applied to select the appropriate deallocation function.

A declaration of placement `operator delete` matches the declaration of a placement `operator new` if it has the same number of parameters and, after parameter transformations (8.3.5), all parameter types except the first are identical. If the deallocation function selected by overload resolution does not match the placement `operator new` called by the *new-expression* or if the lookup for the deallocation function was ambiguous, no deallocation function will be called and the memory in which the object was being constructed will not be freed before the exception is propagated.

In the same section, delete paragraph 21.

In section 5.3.5 [expr.delete], delete paragraph 9 and change the last sentence of paragraph 8 from:

> When the keyword `delete` in a *delete-expression* is preceded by the unary **`::`** operator, the global deallocation function is used to deallocate the storage.

to:

> Selection of the deallocation function to be called when deleting a class object or array thereof is described in 12.5. For non-class objects and arrays of non-class objects, if the keyword `delete` in a *delete-expression* is preceded by the unary **`::`** operator, the deallocation function's name is looked up in the global scope; otherwise, the name is looked up in the context of the *delete-expression* following the normal rules for name lookup (3.4.1), ignoring member functions that might be found in lexically-enclosing class scopes. If the result of the lookup is ambiguous or inaccessible, or if the lookup selects a placement deallocation function, the program is ill-formed.

In section 12.5 [class.free], delete paragraphs 1-3 and replace paragraphs 7 and 8 with the following:

> If a *delete-expression* begins with a unary **`::`** operator, the deallocation function's name is looked up in the global scope. Otherwise, if the *delete-expression* is used to deallocate an object whose static type has a virtual destructor, the deallocation function is the one found by the lookup at the definition of the dynamic type's virtual destructor (12.4). *[Footnote: A similar lookup is not needed for the array version of the delete operator because 5.3.5 requires that in this situation, the static type of the delete-expression-s operand be the same as its dynamic type.]* Otherwise, if the *delete-expression* is used to deallocate an object of class `T` or array thereof, the static and dynamic types of the object shall be identical and the deallocation function's name is looked up in the scope of `T`. If this lookup fails to find the name, the name is looked up in the context of the *delete-expression* following the normal rules for name lookup (3.4.1), ignoring member functions that might be found in lexically-enclosing class scopes. If the result of the lookup is ambiguous or inaccessible, or if the lookup selects a placement deallocation function, the program is ill-formed.