Doc No:	**X3J16/92-0093**
	WG21/N0170
Date:	**18-Sep-92**
Project:	**Programming Language C++**
Ref Doc:	**X3J16/92-0055 / WG21/N0132**
	X3J16/91-0124 / WG21/N0057
	X3J16/92-0012 / WG21/N0090
Reply to:	**Laura Yaker**
	laura_yaker@mentorg.com

Array *new* and *delete* Revisited (revised)

Laura Yaker
laura_yaker@mentorg.com
(503) 685-7000 x 2020

ABSTRACT

This paper is a revision to X3J16/92-0044,WG21/N0132 which proposed a solution to problems related to allocating and deallocating arrays of objects. This revision incorporates information from discussions of the Extensions Working Group at the Toronto (7/92) meeting. The only functional change to the proposal was to change the operator names from `[]new` and `[]delete` to `new[]` and `delete[]`.

Overview

This document is divided into the following sections:

1 Background and Rational
This section describes the reasons for the proposal.

2 Proposal
This section gives an overview of the features of the proposal.

3 Environmental Impact
This section describes the effect of the proposal on existing C++ code, existing implementations, ease of use, the syntax of C++, and the text in the draft standard. (This is the section to look at if you are looking for detailed information.)

4 Summary
This section summarizes the pros and cons of the proposal.

5 Appendix - Alternatives
This section describes the alternatives that were considered.

1 Background and Rationale

Currently arrays of class objects are allocated/deallocated with global operators `new()` and `delete()` even when class specific forms of operators new() and delete() are defined. This behavior presents several problems for class and library implementors.

C++ programmers can control the allocation and deallocation class objects by defining their own member `new` and `delete` operators. They are not, however, able to provide similar control for arrays of class objects. (see papers X3J16/91-0124,WG21/N0057 and X3J16/920012,WG21/N0090 for more information.)

Some tasks that programmers are prevented from doing with arrays are the following:

- placing them on class-specific heaps
- detecting attempts to delete individual members of an array
- monitoring for heap leaks, including primitive garbage collection

2 Proposal

2.1 Description

This section gives an overview of the proposal. For specific details, see section 3.

Two new operators, `operator new[]()` and `operator delete[]()` are proposed for array allocation and deallocation. Global versions of these operators would be provided in the standard C++ library. Users would be able to override these versions with their own and also provide class specific forms of these operators.

- global `operator new[]()`:
  ```
  void * operator new[] (size_t)
  ```
- class `operator new[]()`:
  ```
  void * X::operator new[] (size_t)
  ```
- global `operator delete[]()`:
  ```
  void operator delete[](void *)
  ```
- class `operator delete[]()`:
  ```
  void X::operator delete[](void *)
        or
  void X::operator delete[](void *, size_t)
  ```

These operators would be used whenever arrays were allocated and deallocated where currently the global forms of `new()` and `delete()` are called.

`Operator new[]()` would be able to be overloaded in the same manner as `operator new()`. Additional arguments would be passed to `operator new[] ()` via the ***placement*** syntax.

The `size_t` argument to `operator new[] ()` specifies the total amount of space to be allocated for the array.

Multi-dimensional arrays are handled in the same fashion as single-dimensional arrays with respect to calling operator new[]. (In other words, for multi-dimensional arrays the size argument is still the total space required for the entire array.)

`Operator delete[]()`, like `operator delete()`, would not be overloadable. In its class form, `X::operator delete[]()` would be supplied in either (but not both) the one argument or two argument form.

Like operators `new()` and `delete()`, if a programmer defines `operator new[]()` the programmer is not required to define a corresponding `operator delete[]()`. In most cases, however, programmers will want to define both since memory allocation and deallocation behaviors are often paired.

The presence of `operator new[]()` does not change which constructor is called to initialize the elements of the array; they are still initialized with a default constructor.

2.1.1 Rules for selecting `operator new[]()`

The global `operator new[]()` would be used for allocation of arrays of nonclass types and for arrays of class types if no `operator new[]()` existed for the class. If `operator new[]` existed for the class it would be used for allocation of arrays of that class. Global `void * operator new[](size_t)` would be supplied in the standard C++ library.

Existing member function lookup rules apply for selecting a class `operator new[]()`.

Given the following declarations:

```
class X
{
   public:
   void * operator new(size_t size);
   void * operator new[](size_t size);
};

class Y
{ /* ... */ };
```

the following declarations would be handled as follows:

```
X * xp = new X[10];          // Calls X::operator new[]
Y * yp = new Y[10];          // Calls global operator new[]
X * xp3 = new X[3][5][7];    // Calls X::operator new[]
```

2.1.2 Rules for selecting `operator delete[]()`

The g)bal `operator delete[]()` would be called for deallocating arrays via the existing delet `[]` syntax for arrays of nonclass types and for arrays of class types if no `operator delet []()` existed for the class. If `operator delete[]` existed for the class it would be used)r deallocation of arrays of that class. Global `void operator delete[](void *)` woul be supplied in the standard C++ library.

Exist g member function lookup rules apply for selecting a class `operator delete[]()`.

Give: he following declarations:

```
class X
{
   public:
   void * operator new(size_t size);
   void * operator new[](size_t size);
};

class Y
{ /* ... */ };

X * xp = new X[10];
Y * yp = new Y[10];
X * xp3 = new X[3][5][7];
```

dele g these arrays would be handled as follows:

```
delete [] xp;        // Calls X::operator delete[]
delete [] yp;        // Calls global operator delete[]
delete [] xp3;       // Calls X::operator delete[]
```

2.1. Requirements on the implementations of `operator new[]()` and `delete[]()`

The ly requirement for programmers writing an implementation of `operator new[]()` is th it return a pointer to the storage requested.

The ıly requirement for programmers writing an implementation of `operator dele ![]()` is that it reclaim the space pointed to by the `void *` argument. If the single argu ent form of `operator delete[]()` is used, the function must be able to determine the s e of the space to be deleted. (This is no different than for `operator delete()`.)

The ıplementations of `operator new[]()` and `operator delete[]()` provided in the stan ırd library must call `operators new()` and `delete()` respectively in order to leave exist g C++ programs unaffected.

2.2 Examples

For e examples below, assume that we have a memory management system which has

the following classes and functions available (among others):

```
// General heap management class
class heap
{
   public:
      // Create a new heap
      heap();

      // Get rid of the entire heap
      virtual ~heap();

      // Allocate space of size "size" on the heap
      virtual void * allocate(size_t size);

      // Deallocate the space pointed to by ptr
      virtual void deallocate(void * ptr);
      // ...
};

// Specialized high performance heap with restriction that all
// allocation requests must be for objects of the same size.
class uniheap : public heap
{
   public:
      // Create a new uniform heap
      uniheap(size_t size);

      // Get rid of the entire heap
      ~uniheap();

      // Allocate space on the heap. Note: The size argument
      // is ignored!
      void * allocate(size_t size);

      // Deallocate the space pointed to by ptr
      void deallocate(void * ptr);
};
```

Example 1:

For this example, **class x** requires that all instances of **x** be allocated from a specific pool of memory. It uses the basic heap class for its memory management and makes use of class operators **new** and **new[]** to ensure that all dynamic instances of this class are allocated on the same heap.

```
class X
{
   private:
   static heap * X_heap;      // ptr to class-specific heap
   public:
   void * operator new(size_t size)
      {
         if (!X_heap)
         // Create the class-specific heap if it doesn't already exist
            X_heap = new heap;
         // Allocate the object on the class-specific heap.
         return X_heap->allocate(size);
      };

   void * operator new[](size_t size)
      {
         if (!X_heap)
         // Create the class-specific heap if it doesn't already exist
            X_heap = new heap;
         // Allocate the array on the class-specific heap.
         return X_heap->allocate(size);
      };
...
   void operator delete (void * ptr)
      { if (X_heap)
            // Free the object
            X_heap -> deallocate(ptr);};

   void operator delete[](void * ptr)
      { if (X_heap)
            // Free the array
            X_heap -> deallocate(ptr);};
};
heap * X::X_heap = NULL;

main()
{
   X * scalar_x = new X; // calls X::operator new()
   X * array_x = new X [10]; // calls X::operator new[]();
...
   delete scalar_x; // calls X::operator delete();
   delete [] array_x; // calls X::operator delete[]();
}
```

If one tried to write code to do this same thing today, without **operators new[]** and **delete[]**, one solution would be to add a form of global **operator new** that took a **heap *** as one of its arguments. This, however, requires that users of the class specify the name of the class-specific heap when allocating an array of **x** via **new**. The problem with that approach is that it relies on the user remembering to put in the heap argument and that is error prone. In addition, it requires that the user of the class know something specific about the internal implementation of the class. The class ought to be able to keep the its heap pointer private to itself.

Example 2:

In this example, `class Special_object` has several requirements about how its allocation and deallocation are handled.

1 It's a frequently used class of which a very large number of objects are allocated. For performance reasons it needs the fastest, most efficient type of heap that the memory management system provides.

2 To reduce paging, all objects of type `Special_object` need to be as close to each other in memory as possible.

3 `Special_objects` have the particular property that after a known point in the program they are never referenced again. It is therefore desirable to quickly free the memory that they are occupying.

The heap management system presented above provides us with some features that we can use for class `Special_object`:

- Objects allocated within the same heap are close to each other in memory.
- All the memory in a particular heap can be freed quickly with a single call.
- Class uniheap is particularly efficient in terms of space and access time, but it requires that all objects in the heap be the same size.

Using heaps specific to `Special_objects` will take care of requirements 2 and 3. To meet requirement 1, it is decided to use `class uniheap` as the type of heap for scalar allocations of `class Special_object` in order to get the performance benefits.

A `uniheap` cannot be used for arrays of `class Special_object` (because the allocations could be of varying size) but it is still desirable to use a heap specific to `Special_objects` so that these `Special_objects` can be freed all at once and quickly once the program is done with them.

`Operators Special_object::delete` and `Special_object::delete[]` will be made private so that objects cannot be freed individually. (This particular class has no secondary store or any sort of real destruction so there's no need to call its destructor.)

```
class Special_object
{
   public:
      void * operator new(size_t size);
      void * operator new[]();

      // Pointers to heaps for Special_objects
      static heap * Special_object_single_object_heap;
      static heap * Special_object_array_heap;

      Special_object();
      ~Special_object(){};

   private:
      // Both delete and delete[] are private to prevent
      // users from doing individual deallocations.
      void operator delete(void *);
      void operator delete[](void *);

};

void * Special_object::operator new(size_t size)
{
   // use of uniheap requires all allocations to be of same size.
   assert(size = sizeof(Special_object));

   // Create the scalar heap if necessary
   if (!Special_object_single_object_heap)
      Special_object_single_object_heap =
         new uniheap(sizeof(Special_object));

   return Special_object_single_object_heap->allocate(size);
}

void * Special_object::operator new[](size_t size)
{
   // Create the array heap if necessary
   if (!Special_object_array_heap)
      Special_object_array_heap = new heap(location);

   return Special_object_array_heap->allocate(size);
   }
```

```
main()
{
   Special_object * so = new Special_object;
   Special_object * so_array = new Special_object[100];

// allocate many more Special_objects and arrays of Special_objects
// ...
// complete use of Special_objects

   // Free all the Special_objects we allocated by freeing
   // their heaps.
   delete Special_object::Special_object_single_object_heap;
   delete Special_object::Special_object_array_heap;
}
```

3 Environmental Impact

3.1 Effect on Existing C++ Code

As long as the standard library versions of global `operator new[] ()` and `operator delete[] ()` are required to simply call the corresponding scalar versions of `operators new ()` and `delete ()`, the behavior of existing C++ programs is unchanged.

3.2 Effect on Existing Implementations

Existing compiler implementations would have to be modified to call `operators new[]()` and `delete[]()` (global and class-specific) for array allocations and deallocations. They would also have to make modifications to the grammar and semantic processing to accept declarations and definitions of global and class-specific `operators new[] ()` and `delete[] ()`.

Existing library implementations would have to add definitions of global `operator new[] ()` and `operator delete[] ()`. In order to allow existing programs to remain unchanged, the implementation for each of these functions is to call the corresponding scalar version of the operator.

3.3 Effect on ease of understanding and ease of use

Under the current definition of the language, programmers have to learn that global `operators new()` and `delete()` are called for arrays. This is not obvious to novice programmers and takes them by surprise. It has been my experience that even experienced programmers are taken by surprise and require an explanation that justifies what they see as an anomaly.

If the proposal were adopted, those programmers who had already learned the current behavior would need to learn the new behavior. Assuming that they had already understood that arrays were allocated differently than scalar objects, the new rules would not

be more difficult to understand.

Those novice programmers who are not concerned with writing their own forms of **new** and **delete** would not have to learn the new behavior.

3.4 Grammar Changes

The grammar change required would be the addition of **new[]** and **delete[]** to the list of operators in the non-terminal **operator** (See X3J16/92-0060,WG21/N0137 p. 18-8.).

The names **new[]** and **delete[]** for the operators do introduce some ambiguities in the grammar for C++. Because operator names can appear in expressions, there is a syntactic ambiguity with respect to whether the opening bracket is the beginning of a subscript on **operator new** or part of the **operator new[]** function name. (The same issues apply for **operator delete[]**.)

For example:

```
void * p = operator new[10];
```

Syntactically this is legal, even though it is semantically incorrect.

The ambiguity can be resolved with lookahead. This is similar to problems that already exist in the grammar. For example, the same kind of lookahead that can be used to solve a subset of the ambiguity problems with **expression-statements** and **declarations** can the used to resolve the ambiguity problems with **operators new[]** and **delete[]**.

3.5 Modifications to the Draft Standard

If this proposal is accepted, modifications to the standard would need to be made in sections 5.3.3, 5.3.4, 12.5, and the grammar summary. Alternatively, new sections on **new[]** and **delete[]** could be added rather than modifying 5.3.3 and 5.3.4, however, it is probably easier to understand if descriptions of both the scalar and array forms of **new** and **delete** are kept together.

The sections below describe the changes and additions that would need to be made based on the text in document X3J16/92-0060 - WG21/N0137 (June 1992 Draft). Details about exactly what form the changes should take in the draft are left to the Editor.

3.5.1 New

Information below would be part of section 5.3.3 of the standard. Where information would need to be added, I have stated the additional information. For those paragraphs where text would need to be modified I have included both the original and modified text.

The syntax fragment below is exactly as it is specified in the June 1992 draft in section

5.3.3.

Original:

new-expression:
::$_{opt}$ new-placement$_{opt}$ new-type-id new_initializer$_{opt}$
::$_{opt}$ new-placement$_{opt}$ (type-id)new-initializer$_{opt}$

new-placement:
(expression-list)

new-type-id:
type-specifier-seq new-declarator$_{opt}$

new-declarator:
*** cv-qualifier-seq$_{opt}$ new-declarator$_{opt}$**
qualified-class-specifier :: * cv-qualifier-seq$_{opt}$ new-declarator$_{opt}$
new-declarator$_{opt}$ [expression]

new-initializer:
(expression-list$_{opt}$)

The addition of `operator new[] ()` does not affect this part of the syntax. A form of `operator new[] ()` is called when the new-declarator is **new-declarator$_{opt}$ [expression]**.

The following information would need to be added to section 5.3.3:

Add to/after paragraph 1:

The lifetime of an object created by `operator new[] ()` is not restricted to the scope in which it is created. The new[] operator returns a pointer to the initial element (if any) of the array object. All array dimensions but the first must be constant integral expressions with positive values. The first array dimension can be a general integral expression even when the type-id is used (despite the general restriction of array dimensions in type-ids to constant-expressions). The value of the first array dimension must be non-negative.

Paragraph 2 is unchanged.

Add to/after paragraph 3:

`Operator new[] ()` can be called with the argument zero. In this case a non-null pointer is returned. Repeated such calls return distinct non-null pointers.

Paragraph 4 is unchanged.

Paragraph 5 states:

The new operator will call the function `operator new ()` to obtain storage (*ƒ*12.5). A first argument of `sizeof(T)` is supplied when allocating an object of type T. The ***new-placement*** syntax can be used to supply additional arguments. For example, `new T` results in a call of `operator new (sizeof(T))` and `new(2,f) T` results in a call `operator new (sizeof(t), 2, f)`.

Paragraph 5 should be modified to state:

The new operator will call either the function `operator new ()` or the function `operator new[] ()` to obtain storage (*f*12.5). When **new-declarator**$_{opt}$ [**expression**] (array allocation) is specified the function `operator new[] ()` will be called otherwise, `operator new ()` is called. When `operator new()` is called, a first argument of `sizeof(T)` is supplied when allocating an object of type T. The ***new-placement*** syntax can be used to supply additional arguments. For example, `new T` results in a call of `operator new (sizeof(T))` and `new(2,f) T` results in a call of `operator new (sizeof(T), 2, f)`. When `operator new[] ()` is called, a first argument of `sizeof(AT)` is supplied where `AT` is an array of the specified type and dimensions. The ***new-placement*** syntax can be used to supply additional arguments. For example, `new T[10]` results in a call of `operator new[] (sizeof(T[10]))` and `new(2,f) T[10]` results in a call of `operator new[] (sizeof(T[10]), 2, f)`. (`sizeof(T[10])` is equal to `(sizeof(T) * 10)`.)[1]

Add to/after paragraph 6:

The placement syntax can be used only if an `operator new[] ()` with suitable argument types has been declared.

Paragraph 7 states the following:

When an object of a nonclass type (including arrays of class objects) is created with operator new, the global `::operator new()` is used. When an object of class T is created with operator new, `T::operator new()` is used if it exists (using the usual lookup and access rules for finding members of a class and its base classes; 10.1.1, 11.1); otherwise the global `::operator new()` is used. Using `::new` ensures that the global `::operator new()` is used even if `T::operator new()` exists.

Paragraph 7 should be modified to state:

When an object of a nonclass type, not including arrays of class objects, is created with operator new, the global `::operator new()` is used. When an object of class T is created with operator new, `T::operator new()` is used if it exists (using the usual lookup and access rules for finding members of a class and its base classes; 10.1.1, 11.1); otherwise the global `::operator new()` is used. Using `::new` ensures that the global `::operator new()` is used even if `T::operator new()` exists. When an array of objects of class T is created with operator new, `T::operator new[] ()` is used if it exists (using the usual lookup and access rules for finding members of a class and its base classes; 10.1.1, 11.1); otherwise the global `::operator new[] ()` is used. This applies for both single and multi-dimensional arrays.

Paragraph 8 is unchanged.

Paragraph 9 states the following:

If a class has a constructor an object of that class can be created by **new** only if suitable arguments are provided or if the class has a default constructor (f12.1). Whether the memory for an

---

1. Note that it is not, according to the current draft, legal for the compiler to request any additional space for the array. (See sections 5.3.2 and 5.3.3 on **sizeof(array)** and the value of the first argument to **operator new()**.) However, in a survey of 3 currently available C++ systems, 2 of the 3 asked for more space for the array than **(number-of-elements * size-of-an-element)**.When this was discussed at the Toronto meeting the consensus was that this was an oversight in the draft and that compilers are allowed to request additional space for the array. Assuming that the definition or **sizeof(array)** in 5.3.2 remains the same, then the text describing the size argument to **operator new[]** needs to be clarified to explain exactly what the size is. (Note that this change needs to be made to correctly describe global **operator new** if this proposal is not adopted.)

object of a class with a constructor is allocated before entering the constructor or by the constructor itself is unspecified. In either case, the memory is allocated by `operator new`. Access and ambiguity control are done for both `operator new()` and the constructor; see *f*12.

Paragraph 9 should be modified to say:

If a class has a constructor an object of that class can be created by `new` only if suitable arguments are provided or if the class has a default constructor (f12.1). Whether the memory for an object of a class with a constructor is allocated before entering the constructor or by the constructor itself is unspecified. In either case, the memory is allocated by `operator new` or `operator new[]`. Access and ambiguity control are done for both `operator new()` (or `operator new[] ()`) and the constructor; see *f*12.

Paragraph 10 states the following:

No initializers can be specified for arrays. Arrays of objects of a class with constructors can be created by `operator new` only if the class has a default constructor (*f*12.1). In that case, the default constructor will be called for each element of the array.

Paragraph 10 should be modified to say:

No initializers can be specified for arrays. Arrays of objects of a class with constructors can be created by `operator new[] ()` only if the class has a default constructor (*f*12.1). In that case, the default constructor will be called for each element of the array.

Add to/after paragraph 11:

Any form of `operator new[] ()` may indicate failure to allocate storage by returning 0 (the null pointer). In this case no initialization is done and the value of the expression is 0.[2]

Add to/after paragraph 12:

The order of evaluation of the call to an `operator new[] ()` to get memory and the evaluation of arguments to constructors is unspecified. It is also unspecified if the arguments to a constructor are evaluated if `operator new[] ()` returns 0.

Paragraphs 13-14 are unchanged.

3.5.2 Delete

The information below would be part of section 5.3.4 of the standard. Where information would need to be added, I have stated the additional information. For those paragraphs where text would need to be modified I have included both the original and modified text.

The following is the grammar fragment from section 5.3.4. It is unchanged by the addition of `operator delete[] ()`.

---

2. `Operator new[] ()` should have the same requirements on it as new with respect to returning 0 vs. throwing an exception and ***new_handler***. In the 6/92 Draft, however, it specifies that new may return zero and does not mention either new throwing an exception or setting up a ***new_handler***.

delete-expression:
::$_{opt}$ delete cast-expression
::$_{opt}$ delete [ ] cast-expression

When **delete-expression** evaluates to the **::opt delete [ ] cast-expression** form, an `operator delete[] ()` is called. The result of the expression still has type void.

Paragraphs 1 through 6 remain unchanged.

Paragraph 7 states:

> To free the storage pointed to, the delete operator will call the function `operator delete()`; see *f*12.5. For objects of a nonclass type (including arrays of class objects), the global `::operator delete()` is used. For an object of a class `T`, `T::operator delete()` is used if it exists (using the usual lookup rules for finding members of a class and its base classes (*f* 10.1.1); otherwise the global `::operator delete()` is used. Using `::delete` ensures that the global `::operator delete()` is used even if `T::operator delete()` exists.

Paragraph 7 should be modified to state:

> To free the storage pointed to, the delete operator will call the function `operator delete()` or `operator delete[] ()`; see *f*12.5. For objects of a nonclass type, not including arrays of class objects, the global `::operator delete()` is used. For an object of a class `T`, `T::operator delete()` is used if it exists (using the usual lookup rules for finding members of a class and its base classes; *f*10.1.1); otherwise the global `::operator delete()` is used. Using `::delete` ensures that the global `::operator delete()` is used even if `T::operator delete()` exists. When an array of objects of class T is destroyed with operator delete, `T::operator delete[] ()` is used if it exists (using the usual lookup rules for finding members of a class and its base classes; *f*10.1.1), otherwise the global `::operator delete[] ()` is used.

Paragraphs 8 and 9 remain unchanged.

Paragraph 10 states:

> The default global `operator delete` reclaims storage allocated by the default global `operator new` (only). If other operators `new` or `delete` are invoked, it is the responsibility of the programmer to see that storage is correctly allocated and reclaimed.

Paragraph 10 should be modified to say:

> The default global `operator delete` reclaims storage allocated by the default global `operator new` (only) and the default global `operator delete[]` reclaims storage allocated by the default global `operator new[]` (only). If other operators `new` or `delete` are invoked, it is the responsibility of the programmer to see that storage is correctly allocated and reclaimed.

3.5.3 Free Store

Information below would be part of section *f*12.5 of the standard. Where information would need to be added, I have stated the additional information. For those paragraphs where significant text would need to be modified I have included both the original and modified text.

Paragraph 1: Replace "`operator new()`" with "`operator new()` or `operator new[]()`".

Paragraph 2 states:

An `X::operator new()` for a `class X` is a static member (even if not explicitly declared static). Its first argument must be of type `size_t`, an implementation-dependent integral type defined in the standard header `<stddef.h>`; it must return `void *`. For example,

```
class X {
   // ...
   void* operator new(size_t);
   void* operator new(size_t, Arena *);
};
```

Paragraph 2 should be modified to state:

`X::operator new()` and `X::operator new[]()` for `class X` are static members (even if not explicitly declared static). The first argument of each must be of type `size_t`, an implementation-dependent integral type defined in the standard header `<stddef.h>`; it must return `void *`. For example,

```
class X {
   // ...
   void* operator new(size_t);
   void* operator new(size_t, Arena *);

   void* operator new[](size_t);
   void* operator new[](size_t, Arena *);
};
```

Paragraph 3: Change "an `operator new()`" to "`operator new()` and `operator new[]()`".

Add to/after Paragraph 4:

An `X::operator delete[]()` for a `class X` is a static member (even if not explicitly declared *static*) and must have its first argument of type `void *`; a second argument of type `size_t` may be added. It cannot return a value; its return type must be `void`. For example,

```
class X {
   // ...
   void operator delete[](void *);
};

class Y {
   // ...
   void operator delete[](void *, size_t);
};
```

Add to/after paragraph 5:

Only one `operator delete[]()` may be declared for a single class; thus `operator delete[]()` cannot be overloaded. The global `operator delete[]()` takes a single argument of type `void*`.

Add to/after paragraph 6:

If the two argument style is used, `operator delete[]()` will be called with a second argument indicating the size of the array object being deleted. The size passed is determined by the

destructor (if any) or by the (static) type of the pointer being deleted; that is, it will be correct either if the type of the pointer argument to the delete operator is the exact type of the object (and not, for example, just the type of base class) or if the type is that of a base class with a virtual destructor.

Replace paragraph 7 with:

The global and class forms of **operator new[] ()** and operator delete[] are used for arrays of nonclass and class objects *(f*5.3.3, *f*5.4.4*)*.

Paragraph 8 states the following:

Since **X::operator new ()** and **X::operator delete ()** are **static** they cannot be **virtual**. A destructor finds the **operator delete ()** to use for freeing store using the usual scope rules. For example,

```
struct B {
   virtual ~B();
   void* operator new(size_t);
   void operator delete(void*);
};

struct D : B {
   ~D();
   void* operator new(size_t);
   void operator delete(void*);
};

void f()
{
   B* p = new D;
   delete p;
}
```

Here, storage for the object of class D is allocated by **D::operator new ()** and, thanks to the virtual destructor, deallocated by **D::operator delete ()**. Access to **operator delete ()** is checked as if it were virtual. Thus even though a different one may actually be executed, the statically visible **operator delete ()** must be accessible. In the example above, if **B::operator delete ()** had been **private**, the delete expression would have been illegal.

Replace paragraph 8 with:

Since **X::operator new (), X::operator new[](), X::operator delete (),** and **X::operator delete[] ()** are **static** they cannot be **virtual**. A destructor finds the **operator delete ()** or **operator delete[] ()** to use for freeing store using the usual scope rules. For example,

```
struct B {
   virtual ~B();
   void* operator new(size_t);
   void* operator new[](size_t);
   void operator delete(void*);
   void operator delete[](void*);
};

struct D : B {
   ~D();
   void* operator new(size_t);
   void* operator new[](size_t);
   void operator delete(void*);
   void operator delete[](void*);
};

void f()
{
   B* p = new D;
   B* pa = new D[10];
   delete p;
   delete [] pa;
}
```

Here, storage for scalar objects of class D is allocated by `D::operator new ()` and, thanks to the virtual destructor, deallocated by `D::operator delete ()`. Similarly, storage for an array of class D is allocated by `D::operator new[] ()` and, thanks to the virtual destructor, deallocated by `D::operator delete[] ()`. Access to `operator delete ()` and to `operator delete[] ()` is checked as if it were `virtual`. Thus even though a different one may actually be executed, the statically visible `operator delete ()` and `operator delete[] ()` must be accessible. In the example above, if `B ::operator delete ()` had been `private`, the `delete p` expression would have been illegal. If `B ::operator delete[] ()` had been `private`, the `delete [] pa` expression would have been illegal.

4 Summary

The proposed `operators new[]` and `delete[]` provide solutions for real problems in real programs. They provide a way to control and tracking/monitoring of class object array allocations and deallocations. The new operators also provide programmers the ability to disable dynamically allocated arrays by making `operator new[] private`.

The proposed operators give programmers control of their array allocations and deallocations without exposing implementation-dependent characteristics of the compiler or run-time library. Programmers do not need to know how compilers manage any overhead for arrays such as how they keep track of the number of elements in an array.

The "environmental impact" of this proposed extension is not large. Existing programs would continue to work unchanged. The changes needed in existing compilers and libraries are straightforward.

The use and behavior of the proposed operators is at least as easy to understand as the

current behavior and, may in fact be significantly easier to understand.

5 Appendix - Alternatives

The papers X3J16/91-0124- WG21/N0057 and X3J16/92-0012 /-WG21/N0090 proposed two alternative solutions to the problem of array allocation and deallocation. In addition to those proposals the following alternatives were considered:

- Status quo
  One option is to not change the language at all. In that case, what can programmers do to cope with problem?

  One option is to not use arrays for classes where the allocation and deallocation must be controlled at the class level, but to instead use another mechanism, such as templates to create vectors of class objects. If a programmer has control over both the definition of the class and the code that uses the class, the avoidance of arrays can be handled by programming convention. It is frequently not the case, however, that the producer of the class can rely on the programming conventions of the consumers. For that case there needs to be a way to prevent consumers of a class from declaring arrays of that class.

  It is possible to prevent heap-based arrays of a class from being used by making the default constructor for that class private. Because the default constructor is used to initialize the array, that prevents a consumer of a class from saying:

```
class X
{
...
private:
X(){};
}
X * x = new X[10];
```

  Note that it is still possible for consumers of `class X` to declare stack-based arrays by saying:

```
X x[10];
```

  If the language were ever extended so that constructors other than the default constructor could be used for initialization of heap-based arrays, this trick of making the default constructor private would not be sufficient to disable arrays. It is also important to note that making the default constructor for a class private, can make use of the class more awkward because the default constructor cannot be used at all.

  Creating a general mechanism to handle arrays (including multi-dimensional arrays and use of the array[sub1][sub2][sub3] syntax) and allows the allocation/deallocation of arrays to be handled in a class specific manner is somewhat difficult. For

example, in Lippman's "C++ Primer"[3] there is an example of an array template. That template does not solve the problems that this proposal is intended to address, however, because it uses arrays in its underlying implementation. The calls to `new` of those arrays have the same problems with lack of class specific control as code that uses built-in arrays directly.

While it appears to be possible to write a separate mechanism that does not use arrays as its underlying implementation, it is both cumbersome, more difficult to use and less efficient than built-in arrays. The difficultly in use arises from the fact that users of the mechanism have to be educated in a new form of arrays that is different from the one built into the language. The lack of efficiency arises from the layers of user code that have to be added and at least one additional level of indirection in each pseudo-array.

- An Operation-disabling mechanism
  This alternative arose out of discussions about how to disable arrays. The common method in C++ for "turning off" an operation on a class is to make the appropriate function private. For example, if objects of `class X` should never be allocated on the heap, one can make a private `X::operator new()`. (Note that a user would still be able to create an array of `X`'s on the heap!) For some operations, however, there is not appropriate function to make private. Consider the case of a class which must only be allocated on the heap. There is no function to make private that prevents users from declaring `X x;` and still allows them to declare `X * x = new X`.

  As noted in the alternative described above, it is possible to prevent the creation of an array of a class on the heap by making the default constructor for that class private. It would be cleaner if there were a way to specifically state that arrays were to be disabled (since the connection between the default constructor and arrays is rather tenuous). If there were a general mechanism for disabling operations on a class, that mechanism could be used to disable arrays.

  Unfortunately, this author has not been able to come up with a proposal for solving this general problem without substantial extensions to the language.

- Additional parameter on `new[]` - element count
  A variation on the proposal presented in this paper would be to supply an additional argument to `operator new[]()` that specified the number of elements in the array. That would make the `operator new[]()` signature the following:
  `void * operator new[] (size_t total_size, size_t`[4] `number_of_elements);`

---

3. Lippman, *C++ Primer*, 2nd ed., Addison-Wesley, 1991, pp.380-388
4. See paper X3J16/91-0124 - WG21/N0057 for an explanation of why the type for the second argument should be size_t.

This information was provided to the array form of `new` proposed in X3J16-91-0124-WG21/N0057. For particular applications that need to specifically know the number of elements in the array this form of `operator new[] ()` would provide them with that information. In the original proposal, however, this information had to be provided to array operator new because it was the array operator new's responsibility to keep track of the number of elements for array operator delete. In the extension proposed in this paper, the job of keeping track of the number of elements is done by the compiler.

The additional information about the number of elements to be allocated originally sounded like something that would be useful. Members of the Extensions Working Group, however, have not found an example of how the information would be used that adequately justifies increasing the complexity of the proposal by adding the additional argument or that did not depend upon or involve exposing implementation specific details about arrays.