Proposing Free Store Operators Specifically Devoted to Arrays of Objects

Jim Howard

1. Proposal:

1 Additional variants of operators `new` and `delete` are proposed which are specifically devoted to serving requests related to free store for arrays `[]` of objects:

   ```c
   void *operator new(size_t nitems, size_t item_size);
   ```

   - This signature is inspired by that of the ANSI C function `malloc()` and uses `size_t` for both formal parameters for the same reason.

   ```c
   void operator delete(size_t item_size, void *p);
   ```

   - This signature, heretofore illegal in the language, should be safely definable to this new purpose. The `size_t` argument is largely intended as a signature-distinguishing flag, but may aid some recordkeeping.

2 A corollary of this proposal is that the global operators:

   ```cpp
   ::operator new(size_t)
   ::operator delete(void *)
   ```

   would now be employed only for operations on free store of scalar objects, and no longer perform array allocations.

   - As the language requires that a user specify when an array is deleted, there is no problem in recognizing which variant of `delete` should be called.

3 Additionally, it is proposed that these operators `new` and `delete` be permitted as static members of classes. They would have scope, visibility and disambiguation rules similar to those of other member free store operators permitted in today's language. Thus, user source code for allocations such as:

   ```c
   X *x ay = new X[10];
   ```

   would call the member allocator, if it existed:

   ```c
   X::operator new(10, sizeof(X));
   ```

   and otherwise call:
::operator new(10, sizeof(X));
Similarly, deallocations like:
    delete [] x_ay;
would call the member deallocator, if it existed:
    X::operator delete(sizeof(X), &x_ay[0]);
and otherwise call:
    ::operator delete(sizeof(X), &x_ay[0]);

4 It is also proposed that the placement syntax permitted today for scalar allocators be permitted with
array allocators. For example, assuming the appropriate type match for heap_p:
    X *x_ay = new (heap_p) X[10];
would call, if it exists:
    X::operator new(10, sizeof(X), heap_p);
and otherwise call:
    ::operator new(10, sizeof(X), heap_p);

2. Rationale:

1 The need to carefully manage memory associated with large data structures has led to enhancing the
C++ language to allow operator new() to be a static member and to accept additional
arguments - n.b. placement specifications. The C++ language well supports all of these kinds of
memory allocation for many language-primitive aggregations of user-defined types - structs,
unions, etc. These prior enhancements permit a user-defined type to observe its allocations and to
more carefully control memory layout and reclamation. The proposal extends such support to the
remaining language-primitive aggregation: [] arrays.

2 Defining variants of operators new and delete specifically devoted to [] array allocation and
deallocation is prompted by 3 goals:

- permit objects to observe and control their memory allocations, as is allowed for scalar
  allocations of user-defined types
- provide focus for - and take advantage of - the language requirement to specify when an array is
  being deleted
- specify the "contract" between a compiler and the runtime library regarding allocation of arrays

3 The language, as specified prior to this proposal, offers no way to meet these goals.

4 Array-devoted operators new and delete are provided as a matched pair so that recordkeeping
saved by
    operator new(size_t, size_t)
has a well-defined, exact recipient: its matching
    operator delete(size_t, void *).

3. Discussion:

1 There are three constituents of constructing an array of objects:

   1. an identified function with defined responsibility to allocate memory for the array

Last revised: 7 Nov 91
2. an initialization mechanism to be used for each object in the array
3. an iterator to walk through the allocated memory, employing the initializer to construct each
   object in the array

This proposal addresses 1. above, while 2. is already defined by the language specification as the
object’s default constructor. The iterator (3.) may be an appropriate subject for another proposal.

2 Destroying an array of objects is analogous. We specify a mechanism devoted to storage
deallocation. Individual object tear-down is already defined as the object’s destructor. The implied
iterator is an appropriate topic for another proposal.

3 The "Annotated C++ Reference Manual" expresses (in commentary within §5.3.3) the interpretation
that "an array of type X isn’t an X". While that is valid, that should not prohibit objects of type X
from volunteering an allocator to be used when making an array of its kind. This proposal defines a
way for an object to volunteer such an allocator. If none is volunteered, the appropriate global
allocator is used largely as before.

4 The existing language specification states in §5.3.3 that the global ::operator new() is
employed to allocate memory for an array of objects. Thus, the C++ compiler computes its requested
allocation amount based on the object size and the quantity of objects. However, the compiler is free
to add to that size by any amount it desires. At least one current implementation relies upon the
compiler to add enough extra space to record information about the array allocation to support later
delete operations. This constitutes a hidden contract between the compiler and the implementations
of new and delete. Such a hidden contract is undesirable because it inseparably ties a compiler to
a particular library implementation architecture. If that implementation architecture is not public, we
effectively prevent supplying alternate library implementations as might be desired to address
performance, debugging or testing issues. Devoting specific new and delete operators to array
allocations allows the library to implement a new and delete pair which have a contract between
themselves, but no longer require the compiler’s participation.

5 Benefits gained with the proposal:

✓ avoids here-to-fore hidden contract between a compiler and library provider
✓ provides controllability and observability of array memory allocation in a way similar to
   existing mechanisms for non-arrayed objects
✓ permits use of existing implementations under revised, but now-public interface
✓ increased likelihood that libraries are field-replaceable units
✓ directly supports array allocation recordkeeping required by later delete [] (which
   specifies no element count)

6 Disadvantages of this specific proposal:

× change to existing implementations
× requires specification review for assumptions based on here-to-fore assumed strategy of array
   memory allocation being performed only by the global ::operator new()
× could invalidate existing source where placement variants of operator new() using two
   size_t arguments had been declared:
     operator new(size_t item_size, size_t heaven_knows_why);
4. Alternatives and Augmentations:

1 Requiring use of Templates to create array-like aggregations:
   A primary impetus for the proposal is the defined ability of [] to make arrays of user-defined types as well as built-in types. If [] were prohibited from being used to make array-like aggregations of user-defined types, instead requiring the use of parametric type templates to make and use all such array-like aggregations, the need for this proposal would be largely moot.

   An inability to use [] to create arrays of any type-name - whether user-defined or built-in - would be arguably confusing.

2 Extended function matching for compatibility transition:
   For compatibility with older code and libraries, compilers might implement an extension to the function signature lookup proposed. When presented with source code like:
   ```
   X *x ay= new X[10];
   the compiler understands nitems to be 10, and item_size to be sizeof(X). It would check for the two allocators being proposed, then one more. First:
   ```
   ```
   X::operator new(size_t nitems, size_t item_size);
   and then:
   ```
   ```
   ::operator new(size_t nitems, size_t item_size);
   and then, instead of giving up, accept a match against today's global allocator:
   ```
   ```
   ::operator new(size_t);
   after internally doing the appropriate multiplication of nitems*item_size.
   ```

   Such an extension might best be considered a temporary, transitioning measure, with a definite retirement date.

   The compiler would have the opportunity, if taking the final match of ::operator new(size_t), to continue to add some recordkeeping, overhead bytes to any allocation request, as is done today by some implementations.

3 Alternative signature to avoid possible usage collision:
   To avoid even the smallest likelihood of collisions with users who might have existing source containing a "placement" variant of operator new() defined to have two size_t arguments, one might consider proposing
   ```
   void *operator new(unsigned long nitems, size_t item_size);
   ```
   instead. That signature, heretofore illegal in the language, would be safe when measured against existing source. However, it would ignore the thoughts expressed in the ANSI C rationale regarding size_t's use in certain non-obvious places; refer to §3.3.3.4 and §4.10.3.1 in the rationale.