X3J16/96-0183
WG21/N1001

Proposal for Simplified Allocators

Hans Boehm (boehm@mti.sgi.com)
Matt Austern (austern@mti.sgi.com)

The description of allocators in the current draft standard is inconsistent and incomplete. The requirements for allocators are grossly under-specified, and must be deduced from the ways in which allocators are used by other parts of the library; some of these implicit requirements appear to be inconsistent.

The original motivation for allocators was to support multiple memory models. Since the core C++ language only defines a single memory model, however, this goal is not actually achievable. It is impossible, for example, to write a user-defined type that can be substituted for T&. Similarly, the current allocator specifications require user-defined pointers to be convertible to standard pointers. There is no reason to think that the current design of allocators permits any nontrivial use of nonstandard memory models.

The current description of allocators requires extensive changes if it is to be made consistent and useful. This task is especially difficult because the current specification is sufficiently complex that the implications of minor changes are not obvious. Furthermore, due to current compiler restrictions, there is no real experience with allocators as currently specified, much less with corrected versions of the current specification.

We propose to eliminate essentially all of the current allocator complexity, and to replace it by a much simpler design that has been implemented and tested using currently shipping compilers, and that has known utility.

ALLOCATOR REQUIREMENTS (20.1.5)

An allocator type is required to have the following two static member functions.

```
  void* allocate(size_t positive_number_of_bytes) throw(bad_alloc);
  void deallocate(void *non_null_pointer, size_t size_of_first_arg);
```

Allocate() may not return NULL; it may, however, throw bad_alloc.

The argument to allocate() must be nonzero. Similarly, the first argument to deallocate() must be non-NULL: it must be a pointer to memory that was obtained using allocate(). The second argument must be the size that was passed to allocate() when the memory was allocated.

An allocator type is not required to have any non-static members, or to have any accessible constructors.

There is no complexity requirement for allocate() and deallocate(). The requirement of constant-time complexity is a desirable goal, but it cannot be satisfied on standard hardware and OS platforms.

The intent is that both functions are to be made as fast as possible, especially for repeated allocations of small objects: it should be possible to call these functions directly for allocating small objects, without any appreciable performance penalty. In particular, this means that there is no need for a container to maintain its own free list. This simplifies the implementation of portable thread-safe containers. (This appears to have been the intent of the current draft. The intent should be explicitly stated, however, since it has profound implications for client code.)

STANDARD ALLOCATORS (20.4.1)

Every implementation must provide at least the following two allocators:

1. The class alloc allocates memory obtained from operator ::new or malloc. It provides whatever degree of thread-safety is customary in the target environment.

2. The class single_client_alloc is similar to alloc, except it is not intended to be called concurrently by more than one client. In many environments it will be faster than alloc. In some it will be identical.

It is recommended, but not required, that implementations also supply the following two allocators.

3. The class gc_alloc allocates garbage-collectable memory. Access to memory allocated using gc_alloc after the memory has become unreachable (as defined in Stroustrup's "Proposal to Acknowledge that Garbage Collection for C++ is Possible") is undefined. Gc_alloc::deallocate has no effect.

4. The class per_thread_alloc allows concurrent allocation, but may only allow objects deallocated by a thread to only be reused by that thread. This allows per-thread free lists to be maintained.

In addition, the following allocator adapter shall be provided:

```
template<class T, class alloc>
class object_allocator
{
public:
    static T *allocate(size_t n)
                { return 0 == n? 0 : (T*) alloc::allocate(n * sizeof (T));
}
    static T *allocate(void)
                { return (T*) alloc::allocate(sizeof (T)); }
    static void deallocate(T *p, size_t n)
                { if (0 != n) alloc::deallocate(p, n * sizeof (T)); }
    static void deallocate(T *p)
                { alloc::deallocate(p, sizeof (T)); }
};
```

OTHER CHANGES

The functions construct() and destroy(), which are currently member functions of allocators, shall be made global functions, as they were in the original STL specification. The allocator globals in clause 20.4.1.2 [lib.allocator.globals] shall be removed.

Container constructors shall no longer take allocator arguments.

The default allocator argument for containers shall be alloc.

The current (highly incomplete) discussion of operations involving multiple containers with different allocator instances shall be removed from the draft.

UTILITY

Here we consider which of the intended uses of old-style allocators are still possible in this simpler formulation. Note that some of these uses are also not quite possible using the definition of allocators in the draft standard.

0. Many users will want to implement their own containers; provided that the default allocators are fast enough, however, the vast majority of users are not concerned with defining their own allocators. Implementing user-defined containers is far easier with this version of allocators than with any of the other versions.

1. Different allocation strategies, with different tuning hooks and performance tradeoffs, can easily be implemented. No new restrictions are introduced.

2. The new version of allocators does not support user-defined "smart pointer" type (e.g. checked pointers). Allocators as described in the working paper were intended to support this, but, in light of the requirement of convertibility to standard pointers, it is not clear that they actually do.

Similar functionality can often be provided at the container level (e.g. checked vectors) or without altering pointer representations (e.g. conservative garbage collection, Purify, pointer swizzling for persistence). Furthermore the alternate techniques often have performance and/or usability advantages (e.g. they work with third-party precompiled libraries).

3. Allocation from a fixed number of distinct memory regions (e.g. shared memory segments) is also still possible. We simply define one allocator for one region; one useful way to define these allocators is as a single template with a non-type template parameter. Allocation from a variable number of segments is more cumbersome, but it is still feasible in the cases where it is necessary.

ACKNOWLEDGMENTS

We would like to acknowledge Bjarne Stroustrup for bringing the difficulties with allocators to our attention, and Alex Stepanov for many helpful suggestions.