= Allocator (C + +) =

In C + + computer programming , allocators are an important component of the C + + Standard Library . The standard library provides several data structures , such as list and set , commonly referred to as containers . A common trait among these containers is their ability to change size during the execution of the program . To achieve this , some form of dynamic memory allocation is usually required . Allocators handle all the requests for allocation and deallocation of memory for a given container . The C + + Standard Library provides general @-@ purpose allocators that are used by default , however , custom allocators may also be supplied by the programmer .

Allocators were invented by Alexander Stepanov as part of the Standard Template Library (STL) . They were originally intended as a means to make the library more flexible and independent of the underlying memory model , allowing programmers to utilize custom pointer and reference types with the library . However , in the process of adopting STL into the C + + standard , the C + + standardization committee realized that a complete abstraction of the memory model would incur unacceptable performance penalties . To remedy this , the requirements of allocators were made more restrictive . As a result , the level of customization provided by allocators is more limited than was originally envisioned by Stepanov .

Nevertheless , there are many scenarios where customized allocators are desirable . Some of the most common reasons for writing custom allocators include improving performance of allocations by using memory pools , and encapsulating access to different types of memory , like shared memory or garbage @-@ collected memory . In particular , programs with many frequent allocations of small amounts of memory may benefit greatly from specialized allocators , both in terms of running time and memory footprint .

= = Background = =

Alexander Stepanov and Meng Lee presented the Standard Template Library to the C + + standards committee in March 1994 . The library received preliminary approval , although a few issues were raised . In particular , Stepanov was requested to make the library containers independent of the underlying memory model , which led to the creation of allocators . Consequently , all of the STL container interfaces had to be rewritten to accept allocators .

In adapting STL to be included in the C + + Standard Library , Stepanov worked closely with several members of the standards committee , including Andrew Koenig and Bjarne Stroustrup , who observed that custom allocators could potentially be used to implement persistent storage STL containers , which Stepanov at the time considered an " important and interesting insight " .

The original allocator proposal incorporated some language features that had not yet been accepted by the committee , namely the ability to use template arguments that are themselves templates . Since these features could not be compiled by any existing compiler , there was , according to Stepanov , " an enormous demand on Bjarne [Stroustrup] ' s and Andy [Koenig] ' s time trying to verify that we were using these non @-@ implemented features correctly . " Where the library had previously used pointer and reference types directly , it would now only refer to the types defined by the allocator . Stepanov later described allocators as follows : " A nice feature of STL is that the only place that mentions the machine @-@ related types (...) is encapsulated within roughly 16 lines of code . "

While Stepanov had originally intended allocators to completely encapsulate the memory model , the standards committee realized that this approach would lead to unacceptable efficiency degradations . To remedy this , additional wording was added to the allocator requirements . In particular , container implementations may assume that the allocator 's type definitions for pointers and related integral types are equivalent to those provided by the default allocator , and that all instances of a given allocator type always compare equal , effectively contradicting the original design goals for allocators and limiting the usefulness of allocators that carry state .

Stepanov later commented that , while allocators " are not such a bad [idea] in theory (...) [u] nfortunately they cannot work in practice " . He observed that to make allocators really useful , a

change to the core language with regards to references was necessary .
The 2011 revision of the C + + Standard removed the weasel words requiring that allocators of a given type always compare equal and use normal pointers . These changes make stateful allocators much more useful and allow allocators to manage out @-@ of @-@ process shared memory . The current purpose of allocators is to give the programmer control over memory allocation within containers , rather than to adapt the address model of the underlying hardware . In fact , the revised standard eliminated the ability of allocators to represent extensions to the C + + address model , formally (and deliberately) eliminating their original purpose .

= = Requirements = =

Any class that fulfills the allocator requirements can be used as an allocator . In particular , a class A capable of allocating memory for an object of type T must provide the types A : : pointer , A : : const _ pointer , A : : reference , A : : const _ reference , and A : : value _ type for generically declaring objects and references (or pointers) to objects of type T. It should also provide type A : : size _ type , an unsigned type which can represent the largest size for an object in the allocation model defined by A , and similarly , a signed integral A : : difference _ type that can represent the difference between any two pointers in the allocation model .
Although a conforming standard library implementation is allowed to assume that the allocator 's A : : pointer and A : : const _ pointer are simply typedefs for T * and T const * , library implementors are encouraged to support more general allocators .
An allocator , A , for objects of type T must have a member function with the signature A : : pointer A : : allocate (size _ type n , A < void > : : const _ pointer hint = 0) . This function returns a pointer to the first element of a newly allocated array large enough to contain n objects of type T ; only the memory is allocated , and the objects are not constructed . Moreover , an optional pointer argument (that points to an object already allocated by A) can be used as a hint to the implementation about where the new memory should be allocated in order to improve locality . However , the implementation is free to ignore the argument .
The corresponding void A : : deallocate (A : : pointer p , A : : size _ type n) member function accepts any pointer that was returned from a previous invocation of the A : : allocate member function and the number of elements to deallocate (but not destruct) .
The A : : max _ size ( ) member function returns the largest number of objects of type T that could be expected to be successfully allocated by an invocation of A : : allocate ; the value returned is typically A : : size _ type (-1) / sizeof (T) . Also , the A : : address member function returns an A : : pointer denoting the address of an object , given an A : : reference .
Object construction and destruction is performed separately from allocation and deallocation . The allocator is required to have two member functions , A : : construct and A : : destroy , which handles object construction and destruction , respectively . The semantics of the functions should be equivalent to the following :
The above code uses the placement new syntax , and calls the destructor directly .
Allocators should be copy @-@ constructible . An allocator for objects of type T can be constructed from an allocator for objects of type U. If an allocator , A , allocates a region of memory , R , then R can only be deallocated by an allocator that compares equal to A.
Allocators are required to supply a template class member template < typename U > struct A : : rebind { typedef A < U > other ; } ; , which enables the possibility of obtaining a related allocator , parameterized in terms of a different type . For example , given an allocator type IntAllocator for objects of type int , a related allocator type for objects of type long could be obtained using IntAllocator : : rebind < long > : : other .

= = Custom allocators = =

One of the main reasons for writing a custom allocator is performance . Utilizing a specialized custom allocator may substantially improve the performance or memory usage , or both , of the

program . The default allocator uses operator new to allocate memory . This is often implemented as a thin layer around the C heap allocation functions , which are usually optimized for infrequent allocation of large memory blocks . This approach may work well with containers that mostly allocate large chunks of memory , like vector and deque . However , for containers that require frequent allocations of small objects , such as map and list , using the default allocator is generally slow . Other common problems with a malloc @-@ based allocator include poor locality of reference , and excessive memory fragmentation .

A popular approach to improve performance is to create a memory pool @-@ based allocator . Instead of allocating memory every time an item is inserted or removed from a container , a large block of memory (the memory pool) is allocated beforehand , possibly at the startup of the program . The custom allocator will serve individual allocation requests by simply returning a pointer to memory from the pool . Actual deallocation of memory can be deferred until the lifetime of the memory pool ends . An example of memory pool @-@ based allocators can be found in the Boost C + + Libraries .

Another viable use of custom allocators is for debugging memory @-@ related errors . This could be achieved by writing an allocator that allocates extra memory in which it places debugging information . Such an allocator could be used to ensure that memory is allocated and deallocated by the same type of allocator , and also provide limited protection against overruns .

The subject of custom allocators has been treated by many C + + experts and authors , including Scott Meyers in Effective STL and Andrei Alexandrescu in Modern C + + Design . Meyers emphasises that C + + 98 requires all instances of an allocator to be equivalent , and notes that this in effect forces portable allocators to not have state . Although the C + + 98 Standard did encourage library implementors to support stateful allocators , Meyers calls the relevant paragraph " a lovely sentiment " that " offers you next to nothing " , characterizing the restriction as " draconian " .

In The C + + Programming Language , Bjarne Stroustrup , on the other hand , argues that the " apparently [d] raconian restriction against per @-@ object information in allocators is not particularly serious " , pointing out that most allocators do not need state , and have better performance without it . He mentions three use cases for custom allocators , namely , memory pool allocators , shared memory allocators , and garbage collected memory allocators . He presents an allocator implementation that uses an internal memory pool for fast allocation and deallocation of small chunks of memory , but notes that such an optimization may already be performed by the allocator provided by the implementation .

= = = Usage = = =

When instantiating one of the standard containers , the allocator is specified through a template argument , which defaults to std : : allocator < T > :

Like all C + + class templates , instantiations of standard library containers with different allocator arguments are distinct types . A function expecting an std : : vector < int > argument will therefore only accept a vector instantiated with the default allocator .

= = = Enhancements to allocators in C + + 11 = = =

The C + + 11 standard has enhanced the allocator interface to allow " scoped " allocators , so that containers with " nested " memory allocations , such as vector of strings or a map of lists of sets of user @-@ defined types , can ensure that all memory is sourced from the container 's allocator .

= = Example = =