The following rewrites Sections 20.9.5 (The default allocator) and 20.9.14 (C library) of the most recent ISO C++ standard (August 2010) which is available at <http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2010/n3126.pdf>. Your suggestions on this proposal are most welcome.

**DEADLINE IS 15th OCTOBER 2010**

Notes on this proposal:

* The WHOLE POINT of custom STL memory allocators is that they do CUSTOM ALLOCATION. Requiring that they specifically use ::operator new or ::operator delete for allocating and deallocating raw storage is neither necessary nor desirable when the slightly weaker requirement of they being “a means whose effects and results are compatible with” is sufficient.
* Considering the generic & metaprogramming unfriendly design of operators new and delete, the new functions std::New(), std::NewA(), std::Delete() and std::DeleteA() have been introduced. These allocate and construct via a STL allocator instance which affords much greater flexibility in allocation strategy.
* Memory allocators know when newly allocated memory needs to be zeroed or not because they know which pages have been freshly retrieved from the kernel. Right now C++ has no idea if memory is zeroed or not, so it always zeros memory even when it doesn’t have to. The addition of the zerobits flag to the allocator functions allow STL implementations to save on excessive memory zeroing.

## What does this proposal gain for C++?

Suggested implementations for selected std::allocator<type> member functions:

template<class T> typename allocator<T>::pointer allocator<T>::allocate(size\_type n, allocator<void>::const\_pointer hint, bool zerobits)  
{  
 pointer ret;  
 size\_t nbytes = n\*sizeof(T);  
 uintmax\_t flags = zerobits ? M2\_ZERO\_MEMORY : 0;  
 if(n>1 && nbytes>(128\*1024\*(9-sizeof(void \*)))) flags|=M2\_RESERVE\_MULT(sizeof(void \*));  
 if(!(ret=malloc2(nbytes, alignof(T), flags))) throw bad\_alloc();  
 return ret;  
}

template<class T> typename allocator<T>::pointer\* allocator<T>::allocate(pointer ps[], size\_type ns[], size\_type n\_elems, allocator<void>::const\_pointer hint, bool zerobits)  
{  
 pointer\* ret;  
 if(!(ret=independent\_comalloc(ps, ns, n\_elems, zerobits ? M2\_ZERO\_MEMORY : 0))) throw bad\_alloc();  
 return ret;  
}

template<class T> typename allocator<T>::pointer allocator<T>::reallocate(pointer p, size\_type old\_n, size\_type new\_n, bool mayRelocate, allocator<void>::const\_pointer hint, bool zerobits)  
{  
 pointer ret;  
 size\_t newbytes = new\_n\*sizeof(T);  
 uintmax\_t flags = zerobits ? M2\_ZERO\_MEMORY : 0;  
 if(mayRelocate) flags|=M2\_PREVENT\_MOVE;  
 if(new\_n>1 && newbytes>(128\*1024\*(9-sizeof(void \*)))) flags|=M2\_RESERVE\_MULT(sizeof(void \*));  
 if(!(ret=realloc2(p, newbytes, alignof(T), flags)) && mayRelocate) throw bad\_alloc();  
 return ret;  
}

template<class T, class allocator, class... Args> typename allocator<T>::pointer New(Args&&... args)  
{  
 allocator &alloc = return\_static\_allocator<allocator>();  
 allocator::pointer ret = alloc.allocate(1);  
 try  
 {  
 alloc.construct(ret, args);  
 }  
 catch(...)  
 {  
 alloc.deallocate(ret, 1);  
 throw;  
 }  
 return ret;  
}

And what do these changes gain? Here’s how one would modify the Dinkumware (MSVC STL) std::vector<>::reserve() function (lines 745-773) which manages storage expansion to attempt an in-place storage resize before performing a move construction into new storage:

template<class \_Ty, class \_Ax> void vector<\_Ty, \_Ax>::reserve(size\_type \_Count)  
{ // determine new minimum length of allocated storage  
 if (max\_size() < \_Count)  
 \_Xlen(); // result too long  
 else if (capacity() < \_Count)  
 { // not enough room, reallocate  
 pointer \_Ptr = this->\_Alval.reallocate(this->\_Myfirst, capacity(), \_Count);  
 if(!\_Ptr)  
 {  
 \_Ptr = this->\_Alval.allocate(\_Count);

\_TRY\_BEGIN  
 \_Umove(this->\_Myfirst, this->\_Mylast, \_Ptr);  
 \_CATCH\_ALL  
 this->\_Alval.deallocate(\_Ptr, \_Count);  
 \_RERAISE;  
 \_CATCH\_END

size\_type \_Size = size();  
 if (this->\_Myfirst != 0)  
 { // destroy and deallocate old array  
 \_Destroy(this->\_Myfirst, this->\_Mylast);  
 this->\_Alval.deallocate(this->\_Myfirst,  
 this->\_Myend - this->\_Myfirst);  
 }

this->\_Orphan\_all();  
 this->\_Mylast = \_Ptr + \_Size;  
 this->\_Myfirst = \_Ptr;  
 }  
 this->\_Myend = \_Ptr + \_Count;  
 }  
}

**20.9.5 The default allocator [default.allocator]**

namespace std {  
 template <class T> class allocator;  
 // specialize for void:  
 template <> class allocator<void> {  
 public:  
 typedef void\* pointer;  
 typedef const void\* const\_pointer;  
 // reference-to-void members are impossible.  
 typedef void value\_type;  
 template <class U> struct rebind { typedef allocator<U> other; };  
 };

template <class T> class allocator {  
 public:  
 typedef size\_t size\_type;  
 typedef ptrdiff\_t difference\_type;  
 typedef T\* pointer;  
 typedef const T\* const\_pointer;  
 typedef T& reference;  
 typedef const T& const\_reference;  
 typedef T value\_type;  
 template <class U> struct rebind { typedef allocator<U> other; };  
 allocator() throw();  
 allocator(const allocator&) throw();  
 template <class U> allocator(const allocator<U>&) throw();  
 ~allocator() throw();  
 pointer address(reference x) const;  
 const\_pointer address(const\_reference x) const;  
 pointer allocate(size\_type, allocator<void>::const\_pointer hint = 0, bool zerobits = false);  
 pointer\* allocate(pointer ps[], size\_type ns[], size\_type n\_elems, allocator<void>::const\_pointer hint = 0, bool zerobits = false);  
 pointer\* allocate(pointer ps[], size\_type n, size\_type n\_elems, allocator<void>::const\_pointer hint = 0, bool zerobits = false);  
 pointer reallocate(pointer p, size\_type old\_n, size\_type new\_n, bool mayRelocate = is\_pod<T>::value, allocator<void>::const\_pointer hint = 0, bool zerobits = false);  
 void deallocate(pointer p, size\_type n);  
 void deallocate(pointer ps[], size\_type ns[], size\_type n\_elems);  
 void deallocate(pointer ps[], size\_type n, size\_type n\_elems);  
 size\_type max\_size() const throw();  
 template<class U, class... Args> void construct(U\* p, Args&&... args);  
 template <class U> void destroy(U\* p);  
 };

template<class T, class allocator=allocator<T>, class... Args> typename allocator<T>::pointer New(Args&&... args);

template<class T, class allocator=allocator<T>, class... Args> typename allocator<T>::pointer NewA(size\_t n, Args&&... args);

template<class T, class allocator=allocator<T> > void Delete(typename allocator<T>::const\_pointer p);

template<class T, class allocator=allocator<T> > void DeleteA(typename allocator<T>::const\_pointer p);

}

**20.9.5.1** allocator **members [allocator.members]**

1. Except for the destructor, member functions of the default allocator shall not introduce data races (1.10) as a result of concurrent calls to those member functions from different threads. Calls to these functions that allocate or deallocate a particular unit of storage shall occur in a single total order, and each such deallocation call shall happen before the next allocation (if any) in this order.

pointer address(reference x) const;

1. *Returns:* The actual address of the object referenced by x, even in the presence of an overloaded operator&.

const\_pointer address(const\_reference x) const;

1. *Returns:* The actual address of the object referenced by x, even in the presence of an overloaded operator&.

pointer allocate(size\_type n, allocator<void>::const\_pointer hint=0, bool zerobits = false);

1. [ *Note:* In a container member function, the address of an adjacent element is often a good choice to pass for the hint argument. *—end note* ]
2. *Returns:* a pointer to the initial element of an array of storage of size n \* sizeof(T), aligned appropriately for objects of type T, and whose contents are set to all bits zero if zerobits is true. It is implementation-defined whether over-aligned types are supported (3.11).
3. *Remark:* the storage is obtained by a means whose effects and results are compatible with calling ::operator new(std::size\_t) (18.6.1), but it is unspecified when or how often this function is called. The use of hint is unspecified, but intended as an aid to locality if an implementation so desires.
4. *Throws:* bad\_alloc if the storage cannot be obtained.

pointer\* allocate(pointer ps[], size\_type ns[], size\_type n\_elems, allocator<void>::const\_pointer hint = 0, bool zerobits = false);

1. *Returns:* an array of pointers to storages of size ns[i] \* sizeof(T), aligned appropriately for objects of type T, where 0 <= i < n\_elems, and whose contents are set to all bits zero if zerobits is true. It is implementation-defined whether over-aligned types are supported (3.11).
2. *Remark:* this call is functionally equivalent to a for-loop iterating the invocation of allocate(ns[i], hint), so each returned storage can be individually resized and deallocated. It is intended as a batch allocation mechanism.
3. *Remark:* if ps is null on entry, a newly allocated array sized n\_elems of pointers to pointer shall be returned. This array will need to be explicitly deallocated after usage.
4. *Throws:* bad\_alloc if all the storages cannot be obtained. Any partially allocated storages are deallocated before this function exits with an exception.

pointer\* allocate(pointer ps[], size\_type n, size\_type n\_elems, allocator<void>::const\_pointer hint = 0, bool zerobits = false);

1. *Returns:* an array of pointers to storages of size n \* sizeof(T), aligned appropriately for objects of type T, where 0 <= i < n\_elems, and whose contents are set to all bits zero if zerobits is true. It is implementation-defined whether over-aligned types are supported (3.11).
2. *Remark:* this call is functionally equivalent to a for-loop iterating the invocation of allocate(n, hint), so each returned storage can be individually resized and deallocated. It is intended as a batch allocation mechanism.
3. *Remark:* if ps is null on entry, a newly allocated array sized n\_elems of pointers to pointer shall be returned. This array will need to be explicitly deallocated after usage.
4. *Throws:* bad\_alloc if all the storages cannot be obtained. Any partially allocated storages are deallocated before this function exits with an exception.

pointer reallocate(pointer p, size\_type old\_n, size\_type new\_n, bool mayRelocate = is\_pod<T>::value, allocator<void>::const\_pointer hint=0, bool zerobits = false);

1. *Requires:* p shall be a pointer value obtained from allocate() or reallocate(). old\_n shall equal the value passed as the first argument to the invocation of allocate which returned p, or the value passed as new\_n to a successful invocation of reallocate which returned p.
2. *Effects:* In-place resizes (i.e. resizes without pointer relocation) the storage referenced by p from its existing size to new\_n. If mayRelocate is true, the bit contents of the lesser quantity of new size or existing size of the existing storage may be relocated to a new location, and a pointer to that location returned instead (in which case the old pointer becomes invalid). If zerobits is true and new\_n is larger than old\_n, the contents of the storage between old\_n and new\_n is set to all bits zero.
3. *Returns:* a pointer to the initial element of an array of storage of size n \* sizeof(T), aligned appropriately for objects of type T. It is implementation-defined whether over-aligned types are supported (3.11). If mayRelocate is false and the resizing of storage is not possible without relocation, a null pointer is returned – it is expected that implementations will respond to a null pointer return by performing a move construction of the contents into new storage.
4. *Throws:* bad\_alloc if any additional storage being requested cannot be obtained. Does NOT throw any exception if resizing of storage is not possible without relocation.

void deallocate(pointer p, size\_type n);

1. *Requires:* p shall be a pointer value obtained from allocate() or reallocate(). n shall equal the value passed as the first argument to the invocation of allocate which returned p, or the value passed as new\_n to a successful invocation of reallocate which returned p.
2. *Effects:* Deallocates the storage referenced by p.
3. *Remarks:* Uses a means whose effects and results are compatible with calling ::operator delete(void\*) (18.6.1), but it is unspecified when or how often this function is called.

void deallocate(pointer ps[], size\_type ns[], size\_type n\_elems);

1. *Requires:* ps shall be an array sized n\_elems of pointer values obtained from allocate() or reallocate(). ns shall be an array of values equalling the value passed as the first argument to the invocation of allocate which returned the pointer value, or the value passed as new\_n to a successful invocation of reallocate which returned p.
2. *Effects:* deallocates each of the storages referenced by ps[i] where 0 <= i < n\_elems.
3. *Remark:* this call is functionally equivalent to a for-loop iterating the invocation of deallocate(ps[i], ns[i]). It is intended as a batch deallocation mechanism.

void deallocate(pointer ps[], size\_type n, size\_type n\_elems);

1. *Requires:* ps shall be an array sized n\_elems of pointer values obtained from allocate() or reallocate(). n shall equal the value passed as the first argument to the invocation of allocate which returned each ps[n], or the value passed as new\_n to a successful invocation of reallocate which returned the ps[n].
2. *Effects:* deallocates each of the storages referenced by ps[i] where 0 <= i < n\_elems.
3. *Remark:* this call is functionally equivalent to a for-loop iterating the invocation of deallocate(ps[i], n). It is intended as a batch deallocation mechanism.

size\_type max\_size() const throw();

1. *Returns:* the largest value *N* for which the call allocate(N,0) might succeed.

template <class U, class... Args> void construct(U\* p, Args&&... args);

1. *Effects:* ::new((void \*)p) U(std::forward<Args>(args)...)

template <class U> void destroy(U\* p);

1. *Effects:* p->~U()

**20.9.5.2** allocator **globals [allocator.globals]**

template <class T1, class T2> bool operator==(const allocator<T1>&, const allocator<T2>&) throw();

1. *Returns:* true.

template <class T1, class T2> bool operator!=(const allocator<T1>&, const allocator<T2>&) throw();

1. *Returns:* false.

**20.9.14 C Library [c.malloc]**

1. Table 54 describes the header <cstdlib>.

|  |  |  |
| --- | --- | --- |
| Table 54 — Header <cstdlib> synopsis | | |
| **Type** | **Name(s)** | |
| **Functions**: | calloc  free  free2  malloc | malloc2  malloc\_usable\_size  realloc  realloc2 |

1. The contents are the same as the Standard C library header <stdlib.h>, with the following changes:
2. free2(), malloc2(), and realloc2() shall be declared as follows:

namespace std  
{  
 void free2(void \*ptr, uintmax\_t flags = 0);

void \*malloc2(size\_t size, size\_t alignment = 0, uintmax\_t flags = 0);

void \*realloc2(void \*ptr, size\_t size, size\_t alignment = 0, uintmax\_t flags = 0);

}

1. free(), malloc(), and realloc() shall be defined as follows:

namespace std  
{  
 inline void free(void \*ptr, uintmax\_t flags = 0)  
 {  
 free2(ptr, flags);  
 }

inline void \*malloc(size\_t size, size\_t alignment = 0, uintmax\_t flags = 0)  
 {  
 return malloc2(size, alignment, flags);  
 }

inline void \*realloc(void \*ptr, size\_t size, size\_t alignment = 0, uintmax\_t flags = 0)  
 {  
 return realloc2(ptr, size, alignment, flags);  
 }

}

1. The functions calloc(), malloc(), malloc2(), realloc() and realloc2() do not attempt to allocate storage by calling ::operator new() (18.6).
2. The functions free() and free2() do not attempt to deallocate storage by calling ::operator delete().

See also: ISO C Clause 7.11.2.

1. Storage allocated directly with malloc(), malloc2(), calloc(), realloc() or realloc2() is implicitly declared reachable (see 3.7.4.3) on allocation, ceases to be declared reachable on deallocation, and need not cease to be declared reachable as the result of an undeclare\_reachable() call. [ *Note:* This allows existing C libraries to remain unaffected by restrictions on pointers that are not safely derived, at the expense of providing far fewer garbage collection and leak detection options for malloc()-allocated objects. It also allows malloc() to be implemented with a separate allocation arena, bypassing the normal declare\_reachable() implementation. The above functions should never intentionally be used as a replacement for declare\_reachable(), and newly written code is strongly encouraged to treat memory allocated with these functions as though it were allocated with operator new. *—end note* ]
2. Table 55 describes the header <cstring>.

|  |  |  |
| --- | --- | --- |
| Table 55 — Header <cstring> synopsis | | |
| **Type** | **Name(s)** | |
| **Macro**: | NULL | |
| **Type**: | size\_t | |
| **Functions**: | memchr  memmove  memcpy | memcmp  memset |

1. The contents are the same as the Standard C library header <string.h>, with the change to memchr() specified in 21.7.

See also: ISO C Clause 7.11.2.