Allocators in C++11

Alisdair Meredith

Thursday, May 16, 2013

Copyright Notice

Topics to cover

- Motivation
- Bloomberg Allocators
- (The problem with) C++03 allocators
- (The solution for) C++11 allocators
- Experience with C++11 model

What is an allocator?

- mechanism that supplies memory on demand
- typically used as an implementation detail of an object managing state
- typically, but not always, a container

Goal: per-object allocation

- Each object can use the most appropriate allocator, given its context
- A reasonable allocator is supplied by default if the user has no special need

Motivating Examples

- thread-specific allocators
- pooled allocators
- stack-based allocators
- diagnostic / test allocators
- shared-memory allocators

Default Allocator

- The allocator used by default, unless the user supplies their own
- The default Default Allocator is the `NewDelete` allocator
- Default allocator should be set only in `main`
- Typically, a test driver will install a `TestAllocator`

Buffered Sequential Allocator: design

- Initially allocate memory from a supplied buffer
  - typically an array on the stack
- fall back to a second allocator if buffer capacity exceeded
  - typically the default `NewDelete` allocator
- deallocate is a no-op
  - memory reclaimed only when allocator is destroyed

Buffered Sequential Allocator: benefits

- efficient memory allocation
- no contention/synchronization
- low memory fragmentation
- (locality of reference)
- degrades gracefully when over-committed

Buffered Sequential Allocator: risks

- not thread safe
- wasteful for many allocate and release cycles
- poor match for containers that resize in regular use
- best when upper bound of memory consumption is known in advance

Buffered Sequential Allocator: use cases

- short duration containers of a known capacity
  - building a string
  - computing a function over associated values on a small range

Using BDE allocators

```
enum { SIZE = 3 * 100 * sizeof(double) };

char buffer[SIZE] alignas double;

bdlma::BufferedSequentialAllocator alloc(buffer, SIZE);

bsl::vector<double> v1(&alloc);  v1.reserve(100);
bsl::vector<double> v2(&alloc);  v2.reserve(100);
bsl::vector<double> v3(&alloc);  v3.reserve(100);

// do some work...
```

Using BDE allocators

```
enum { SIZE = 3 * 100 * sizeof(double) };

char buffer[SIZE] alignas double;

bdlma::BufferedSequentialAllocator alloc(buffer, SIZE);

bsl::vector<double> v1(&alloc);  v1.reserve(100);
bsl::vector<double> v2(&alloc);  v2.reserve(100);
bsl::vector<double> v3(&alloc);  v3.reserve(100);

// do some work...
```

Using BDE allocators

```
enum { SIZE = 3 * 100 * sizeof(double) };

char buffer[SIZE] alignas double;

bdlma::BufferedSequentialAllocator alloc(buffer, SIZE);

bsl::vector<double> v1(&alloc);  v1.reserve(100);
bsl::vector<double> v2(&alloc);  v2.reserve(100);
bsl::vector<double> v3(&alloc);  v3.reserve(100);

// do some work...
```

Using BDE allocators

```
enum { SIZE = 3 * 100 * sizeof(double) };

char buffer[SIZE] alignas double;

bdlma::BufferedSequentialAllocator alloc(buffer, SIZE);

bsl::vector<double> v1(&alloc);  v1.reserve(100);
bsl::vector<double> v2(&alloc);  v2.reserve(100);
bsl::vector<double> v3(&alloc);  v3.reserve(100);

// do some work...
```

Using BDE allocators

```
enum { SIZE = 3 * 100 * sizeof(double) };

char buffer[SIZE] alignas double;

bdlma::BufferedSequentialAllocator alloc(buffer, SIZE);

bsl::vector<double> v1(&alloc);  v1.reserve(100);
bsl::vector<double> v2(&alloc);  v2.reserve(100);
bsl::vector<double> v3(&alloc);  v3.reserve(100);

// do some work...
```

Vocabulary Types

- Vocabulary types are critical for public APIs
- If allocator is part of type, makes for poor vocabulary
- Allocators supply memory
  - should not depend on object type
- Containers need allocators
  - but allocator should not be part of the type
- Classic example: `bsl::string`

bslma::allocator

```
class Allocator {
  public:
    // PUBLIC TYPES
    typedef bsls::Types::size_type size_type;

    // CLASS METHODS
    static void throwBadAlloc();

    // CREATORS
    virtual ~Allocator();

    // MANIPULATORS
    virtual void *allocate(size_type size) = 0;

    virtual void deallocate(void *address) = 0;

    template <class TYPE>
    void deleteObject(const TYPE *object);

    template <class TYPE>
    void deleteObjectRaw(const TYPE *object);
};

void *operator new(std::size_t                    size,
                   BloombergLP::bslma::Allocator& basicAllocator);

void operator delete(void                            *address,
                     BloombergLP::bslma::Allocator&  basicAllocator);
```

bslma::allocator

```
class Allocator {
  public:
    // ...

    virtual void *allocate(size_type size) = 0;

    virtual void deallocate(void *address) = 0;

    // ...
};
```

bsl::allocator<T>

```
template <class T>
class allocator {
    BloombergLP::bslma::Allocator *d_mechanism;

  public:
    typedef std::size_t     size_type;
    typedef std::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();
    allocator(BloombergLP::bslma::Allocator *mechanism);   // IMPLICIT
    allocator(const allocator& original);
    template <class U>
    allocator(const allocator<U>& rhs);

    pointer allocate(size_type n, const void *hint = 0);
    void deallocate(pointer p, size_type n = 1);

    void construct(pointer p, const T& val);

    void destroy(pointer p);

    BloombergLP::bslma::Allocator *mechanism() const;

    pointer address(reference x) const;
    const_pointer address(const_reference x) const;

    size_type max_size() const;
};
```

BDE Allocators

- Each object can use the most appropriate allocator, given its context
- allocator passed by address
- container does not own the allocator, so user must ensure lifetimes nest

Consequences

- BDE allocators do not 'propagate'
- Copy construction uses the default allocator, if none is supplied
  - Should not return a BDE container by-value
- Cannot 'swap' two BDE containers unless they have the same allocator
- Elements must have same allocator as their container
  - containers must pass allocator to element constructors

Propagating allocators

```
enum { SIZE = 500 * sizeof(string) };

bsls::AlignedBuffer<SIZE> buffer1;
bsls::AlignedBuffer<SIZE> buffer2;

bdlma::BufferedSequentialAllocator a1(buffer1.buffer(), SIZE);
bdlma::BufferedSequentialAllocator a2(buffer2.buffer(), SIZE);

bsl::vector<string> v1(&a1);  v1.reserve(100);
bsl::vector<string> v2(&a2);  v2.reserve(100);

v1.push_back("Hello World");
v2.push_back("Bonjour le monde");

v1.front().swap(v2.front());   // undefined behavior?
swap(v1.front(), v2.front());  // undefined behavior?
```

Propagating allocators

```
enum { SIZE = 500 * sizeof(string) };

bsls::AlignedBuffer<SIZE> buffer1;
bsls::AlignedBuffer<SIZE> buffer2;

bdlma::BufferedSequentialAllocator a1(buffer1.buffer(), SIZE);
bdlma::BufferedSequentialAllocator a2(buffer2.buffer(), SIZE);

bsl::vector<string> v1(&a1);  v1.reserve(100);
bsl::vector<string> v2(&a2);  v2.reserve(100);

v1.push_back("Hello World");
v2.push_back("Bonjour le monde");

v1.front().swap(v2.front());   // undefined behavior?
swap(v1.front(), v2.front());  // undefined behavior?
```

Propagating allocators

```
enum { SIZE = 500 * sizeof(string) };

bsls::AlignedBuffer<SIZE> buffer1;
bsls::AlignedBuffer<SIZE> buffer2;

bdlma::BufferedSequentialAllocator a1(buffer1.buffer(), SIZE);
bdlma::BufferedSequentialAllocator a2(buffer2.buffer(), SIZE);

bsl::vector<string> v1(&a1);  v1.reserve(100);
bsl::vector<string> v2(&a2);  v2.reserve(100);

v1.push_back(“Hello World”);
v2.push_back(“Bonjour le monde”);

v1.front().swap(v2.front());    // undefined behavior?
swap(v1.front(), v2.front());   // undefined behavior?
```

Propagating allocators

```
enum { SIZE = 500 * sizeof(string) };

bsls::AlignedBuffer<SIZE> buffer1;
bsls::AlignedBuffer<SIZE> buffer2;

bdlma::BufferedSequentialAllocator a1(buffer1.buffer(), SIZE);
bdlma::BufferedSequentialAllocator a2(buffer2.buffer(), SIZE);

bsl::vector<string> v1(&a1);  v1.reserve(100);
bsl::vector<string> v2(&a2);  v2.reserve(100);

v1.push_back("Hello World");
v2.push_back("Bonjour le monde");

v1.front().swap(v2.front());    // undefined behavior?
swap(v1.front(), v2.front());   // undefined behavior?
```

Propagating allocators

```
enum { SIZE = 500 * sizeof(string) };

bsls::AlignedBuffer<SIZE> buffer1;
bsls::AlignedBuffer<SIZE> buffer2;

bdlma::BufferedSequentialAllocator a1(buffer1.buffer(), SIZE);
bdlma::BufferedSequentialAllocator a2(buffer2.buffer(), SIZE);

bsl::vector<string> v1(&a1);  v1.reserve(100);
bsl::vector<string> v2(&a2);  v2.reserve(100);

v1.push_back(string("Hello World"));        // which allocator?
v2.push_back(string("Bonjour le monde"));   // which allocator?

v1.front().swap(v2.front());   // undefined behavior?
swap(v1.front(), v2.front());  // undefined behavior?
```

Propagating allocators

```
enum { SIZE = 500 * sizeof(string) };

bsls::AlignedBuffer<SIZE> buffer1;
bsls::AlignedBuffer<SIZE> buffer2;

bdlma::BufferedSequentialAllocator a1(buffer1.buffer(), SIZE);
bdlma::BufferedSequentialAllocator a2(buffer2.buffer(), SIZE);

bsl::vector<string> v1(&a1);  v1.reserve(100);
bsl::vector<string> v2(&a2);  v2.reserve(100);

v1.emplace_back("Hello World");             // BDE 2.18
v2.emplace_back("Bonjour le monde");        // BDE 2.18

v1.front().swap(v2.front());  // undefined behavior?
swap(v1.front(), v2.front());  // undefined behavior?
```

Example allocators

- bslma::NewDeleteAllocator
- bslma::TestAllocator
- bdlma::BufferedSequentialAllocator
- bdlma::MultipoolAllocator
- shared memory allocator?

(The problem with) C++03 allocators

- (or why Bloomberg joined the ISO committee)

(The problem with) C++03 allocators

- Many standard components can use a user-supplied allocator
  - But the allocator forms part of the type
  - Too late to fix this
- Allocator adapters may mitigate this but...
  - C++03 allows implementers to bend the rules
  - simple allocators are still too complex

(The problem with) C++03 allocators

- Many standard components can use a user-supplied allocator
  - But the allocator forms part of the type
  - Too late to fix this
- Allocator adapters may mitigate this but...
  - C++03 allows implementers to bend the rules
  - simple allocators are still too complex

Weasel Words

- An implementation may assume:
  - All instances of a given allocator type are required to be interchangeable and always compare equal to each other.
  - The typedef members `pointer`, `const_pointer`, `size_type`, and `difference_type` are required to be `T*`, `T const*`, `size_t`, and `ptrdiff_t`, respectively.

Weasel Words

- An implementation may assume:
  - All instances of a given allocator type are required to be interchangeable and always compare equal to each other.
- Translation:

Weasel Words

- An implementation may assume:
  - All instances of a given allocator type are required to be interchangeable and always compare equal to each other.
- Translation: allocator objects cannot have state

Weasel Words

- An implementation may assume:
    - The typedef members `pointer`, `const_pointer`, `size_type`, and `difference_type` are required to be `T*`, `T const*`, `size_t`, and `ptrdiff_t`, respectively.
- Translation:

Weasel Words

- An implementation may assume:
  - The typedef members `pointer`, `const_pointer`, `size_type`, and `difference_type` are required to be `T*`, `T const*`, `size_t`, and `ptrdiff_t`, respectively.
- Translation: allocators cannot return smart pointers, such as to shared memory

(The solution is) C++11 allocators

- remove the 'weasel words'
- `allocator_traits` describes the customizable behavior of allocators
  - supplies defaults for majority of interface
- Containers request allocator services through the traits template
  - rather than calling allocator methods directly

Allocator Traits

```
template <class Alloc>
struct allocator_traits {
    typedef Alloc allocator_type;
    typedef typename Alloc::value_type value_type;
    typedef see below pointer;
    typedef see below const_pointer;
    typedef see below void_pointer;
    typedef see below const_void_pointer;
    typedef see below difference_type;
    typedef see below size_type;
    // ...
};
```

Allocator Traits

```
template <class Alloc>
struct allocator_traits {
   // ...

   template <class T>
   using rebind_alloc = see below;

   template <class T>
   using rebind_traits = allocator_traits<rebind_alloc<T>>;

   // ...
};
```

Allocator Propagation

```
template <class Alloc>
struct allocator_traits {
   // ...

   typedef see below propagate_on_container_copy_assignment;
   typedef see below propagate_on_container_move_assignment;
   typedef see below propagate_on_container_swap;

};
```

Allocator Traits

```
template <class Alloc>
struct allocator_traits {
   // ...
   static pointer allocate(Alloc& a, size_type n);
   static pointer allocate(Alloc& a, size_type n, const_void_pointer hint);
   static void  deallocate(Alloc& a, pointer p, size_type n);

   template <class T, class... Args>
   static void construct(Alloc& a, T* p,  Args&&... args);

   template <class T>
   static void destroy(Alloc& a, T* p);

   static size_type max_size(const Alloc& a);
   static Alloc select_on_container_copy_construction(const Alloc& rhs);
  // ...
};
```

Pointer Traits

```
template <class Ptr>
struct pointer_traits {
   typedef Ptr pointer;
   typedef see below element_type;
   typedef see below difference_type;

   template <class U>
   using rebind = see below;

   static pointer pointer_to(see below r);
};
```

Implementing the traits

allocator_traits::size_type

```
template <typename ALLOCATOR>
auto dispatch_size_type(...)
  -> typename ::std::make_unsigned<difference_type<ALLOCATOR>>::type;

template <typename ALLOCATOR>
auto dispatch_size_type(int)
  -> typename ALLOCATOR::size_type;



template <typename ALLOCATOR>
using size_type = decltype(dispatch_size_type<ALLOCATOR>(0));
```

A quick lesson in SFINAE

- Substitution Failure Is Not An Error
- Necessary language feature to support dependent types in function templates
  - ```
    template <class T>
    typename T::type *make_child(T *parent);
    ```
- Discovered this can be (ab)used to control overload resolution in generic code
- Automated in C++11 with `enable_if`
    ```

Worked Example

```
struct true_type  { char dummy[17]; };
struct false_type { char dummy[ 1]; };

template <typename T>
true_type sniff_pointer(typename T::pointer *);

template <typename T>
false_type sniff_pointer(...);

template <typename T>
struct has_pointer {
   static const bool result =
      sizeof(sniff_pointer<T>(0)) == sizeof(true_type);
};
```

Worked Example

```
template <typename T>
struct has_pointer {
   static const bool result =
      sizeof(sniff_pointer<T>(0)) == sizeof(true_type);
};

template <typename Alloc>
struct allocator_traits {
   typedef typename conditional<
            has_pointer<Alloc>::result,
            typename Alloc::pointer,
            typename Alloc::value_type *>::type pointer;
};
```

Worked Example

```
template <typename T>
struct has_pointer {
    static const bool result =
        sizeof(sniff_pointer<T>(0)) == sizeof(true_type);
};

template <typename Alloc>
struct allocator_traits {
    typedef typename conditional<
              has_pointer<Alloc>::result,
              typename Alloc::pointer,
              typename Alloc::value_type *>::type pointer;
};
```

Worked Example

```
template <typename Alloc, bool>
struct default_pointer {
   typedef typename Alloc::value_type *type;
};

template <typename Alloc>
struct default_pointer<Alloc, true> {
   typedef typename Alloc::pointer type;
};

template <typename Alloc>
struct allocator_traits {
   typedef typename
      default_pointer<Alloc,
                      has_pointer<Alloc>::result>::type
      pointer;
};
```

C++11 enables new techniques

- `decltype` expressions
- late specified return types
  - ```
    template <typename T, typename U>
    auto plus(T t, U u) -> decltype(t + u);
    ```
- generalized SFINAE
- alias templates
    ```

Implementing the traits

allocator_traits::size_type

```
template <typename ALLOCATOR>
auto dispatch_size_type(...)
  -> typename ::std::make_unsigned<difference_type<ALLOCATOR>>::type;

template <typename ALLOCATOR>
auto dispatch_size_type(int)
  -> typename ALLOCATOR::size_type;
```

```
template <typename ALLOCATOR>
using size_type = decltype(dispatch_size_type<ALLOCATOR>(0));
```

Implementing the traits

allocator_traits::size_type

```
template <typename ALLOCATOR>
auto dispatch_size_type(...)
  -> typename ::std::make_unsigned<difference_type<ALLOCATOR>>::type;

template <typename ALLOCATOR>
auto dispatch_size_type(int)
  -> typename ALLOCATOR::size_type;

template <typename ALLOCATOR>
using size_type = decltype(dispatch_size_type<ALLOCATOR>(0));
```

Implementing the traits

allocator_traits::size_type

```
template <typename ALLOCATOR>
auto dispatch_difference_type(...)
  -> ::std::pointer_traits<pointer_type<ALLOCATOR>>::difference_type;

template <typename ALLOCATOR>
auto dispatch_difference_type(int)
  -> typename ALLOCATOR::difference_type;

template <typename ALLOCATOR>
using difference_type = decltype(dispatch_difference_type<ALLOCATOR>(0));

template <typename ALLOCATOR>
auto dispatch_size_type(...)
  -> typename ::std::make_unsigned<difference_type<ALLOCATOR>>::type;

template <typename ALLOCATOR>
auto dispatch_size_type(int)
  -> typename ALLOCATOR::size_type;

template <typename ALLOCATOR>
using size_type = decltype(dispatch_size_type<ALLOCATOR>(0));
```

Implementing the traits

allocator_traits::size_type

```
template <typename ALLOCATOR>
auto dispatch_difference_type(...)
  -> ::std::pointer_traits<pointer_type<ALLOCATOR>>::difference_type;

template <typename ALLOCATOR>
auto dispatch_size_type(...)
  -> typename ::std::make_unsigned<difference_type<ALLOCATOR>>::type;

template <typename ALLOCATOR>
using size_type = decltype(dispatch_size_type<ALLOCATOR>(0));
```

Implementing the traits

allocator_traits::size_type

```
template <typename POINTER>
auto pointer_difference_type(...)
  -> decltype((char *)nullptr - (char *)nullptr);

template <typename POINTER>
auto pointer_difference_type(int)
  -> typename ALLOCATOR::difference_type;

template <typename POINTER>
using difference_type = decltype(pointer_difference_type<POINTER>(0));

template <typename ALLOCATOR>
auto dispatch_difference_type(...)
  -> ::std::pointer_traits<pointer_type<ALLOCATOR>>::difference_type;

template <typename ALLOCATOR>
auto dispatch_size_type(...)
  -> typename ::std::make_unsigned<difference_type<ALLOCATOR>>::type;

template <typename ALLOCATOR>
using size_type = decltype(dispatch_size_type<ALLOCATOR>(0));
```

Implementing Traits Functions

```
template <typename TARGET_TYPE, typename ALLOCATOR, typename... ARG_TYPES>
auto do_construct(ALLOCATOR & a, TARGET_TYPE * p, ARG_TYPES &&... args)
  -> decltype((void)a.construct(p, ::std::forward<ARG_TYPES>(args)...))
{
   a.construct(p, ::std::forward<ARG_TYPES>(args)...);
}

template <typename TARGET_TYPE, typename ALLOCATOR, typename... ARG_TYPES>
void do_construct(ALLOCATOR &, void * p, ARG_TYPES &&... args)
{
   ::new (p) TARGET_TYPE( ::std::forward<ARG_TYPES>(args)... );  // not {} initialization
}


template <typename ALLOCATOR>
template <typename TYPE, typename... ARG_TYPES>
void allocator_traits<ALLOCATOR>::construct(ALLOCATOR & a, TYPE * p, ARG_TYPES &&... args)
{
   using PtrType = ::std::add_pointer<typename ::std::remove_cv<TYPE>>::type;

   do_construct<TYPE>(a, const_cast<PtrType>(p), ::std::forward<ARG_TYPES>(args)...);
}
```

Implementing an allocator

```
template <class T>
struct allocator {
   using size_type       = size_t;
   using difference_type = ptrdiff_t;
   using pointer         = T*;
   using const_pointer   = const T*;
   using reference       = T&;
   using const_reference = const T&;
   using value_type      = T;

   template <class U> struct rebind { using other = allocator<U>; };

   allocator() noexcept;
   allocator(const allocator&) noexcept;
   template <class U> allocator(const allocator<U>&) noexcept;
   ~allocator();

   auto address(reference x)       const noexcept -> pointer;
   auto address(const_reference x) const noexcept -> const_pointer;

   auto allocate( size_type, allocator<void>::const_pointer hint = 0) -> pointer;
   void deallocate(pointer p, size_type n);
   auto max_size() const noexcept -> pointer;

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

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

template <class T, class U>
bool operator==(const allocator<T>&, const allocator<U>&) noexcept;
template <class T, class U>
bool operator!=(const allocator<T>&, const allocator<U>&) noexcept;
```

```
template <class T>
struct allocator {
   using size_type       = size_t;
   using difference_type = ptrdiff_t;
   using pointer         = T*;
   using const_pointer   = const T*;
   using reference       = T&;
   using const_reference = const T&;
   using value_type      = T;

   template <class U> struct rebind { using other = allocator<U>; };

   allocator() noexcept;
   allocator(const allocator&) noexcept;
   template <class U> allocator(const allocator<U>&) noexcept;
   ~allocator();

   auto address(reference x)       const noexcept -> pointer;
   auto address(const_reference x) const noexcept -> const_pointer;

   auto allocate( size_type, allocator<void>::const_pointer hint = 0) -> pointer;
   void deallocate(pointer p, size_type n);
   auto max_size() const noexcept -> pointer;

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

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

template <class T, class U>
bool operator==(const allocator<T>&, const allocator<U>&) noexcept;
template <class T, class U>
bool operator!=(const allocator<T>&, const allocator<U>&) noexcept;
```

bslma::allocator

```
class Allocator {
  public:
    // PUBLIC TYPES
    typedef bsls::Types::size_type size_type;

    // CLASS METHODS
    static void throwBadAlloc();

    // CREATORS
    virtual ~Allocator();

    // MANIPULATORS
    virtual void *allocate(size_type size) = 0;

    virtual void deallocate(void *address) = 0;

    template <class TYPE>
    void deleteObject(const TYPE *object);

    template <class TYPE>
    void deleteObjectRaw(const TYPE *object);
};

void *operator new(std::size_t                    size,
                   BloombergLP::bslma::Allocator& basicAllocator);

void operator delete(void                             *address,
                     BloombergLP::bslma::Allocator&  basicAllocator);
```

bsl::allocator<T>

```
template <class T>
class allocator {
    BloombergLP::bslma::Allocator *d_mechanism;

  public:
    typedef std::size_t     size_type;
    typedef std::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();
    allocator(BloombergLP::bslma::Allocator *mechanism);   // IMPLICIT
    allocator(const allocator& original);
    template <class U>
    allocator(const allocator<U>& rhs);

    pointer allocate(size_type n, const void *hint = 0);
    void deallocate(pointer p, size_type n = 1);

    void construct(pointer p, const T& val);

    void destroy(pointer p);

    BloombergLP::bslma::Allocator *mechanism() const;

    pointer address(reference x) const;
    const_pointer address(const_reference x) const;

    size_type max_size() const;
};
```

bsl::allocator<T>

```
template <class T>
class allocator {
    BloombergLP::bslma::Allocator *d_mechanism;

  public:
    typedef std::size_t     size_type;
    typedef std::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();
    allocator(BloombergLP::bslma::Allocator *mechanism);   // IMPLICIT
    allocator(const allocator& original);
    template <class U>
    allocator(const allocator<U>& rhs);

    pointer allocate(size_type n, const void *hint = 0);
    void deallocate(pointer p, size_type n = 1);

    void construct(pointer p, const T& val);

    void destroy(pointer p);

    BloombergLP::bslma::Allocator *mechanism() const;

    pointer address(reference x) const;
    const_pointer address(const_reference x) const;

    size_type max_size() const;
};
```

bsl::allocator<T>

```
template <class T>
class allocator {
    BloombergLP::bslma::Allocator *d_mechanism;

  public:

    typedef T                  value_type;

    allocator();
    allocator(BloombergLP::bslma::Allocator *mechanism);    // IMPLICIT

    template <class U>
    allocator(const allocator<U>& rhs);

    pointer allocate(size_type n, const void *hint = 0);
    void deallocate(pointer p, size_type n = 1);

    void construct(pointer p, const T& val);

    void destroy(pointer p);

    BloombergLP::bslma::Allocator *mechanism() const;

};
```

bsl::allocator<T>

```
template <class T>
class allocator {
    BloombergLP::bslma::Allocator *d_mechanism;

  public:





    typedef T               value_type;









    allocator();
    allocator(BloombergLP::bslma::Allocator *mechanism);   // IMPLICIT

    template <class U>
    allocator(const allocator<U>& rhs);

    pointer allocate(size_type n);
    void deallocate(pointer p, size_type n);
    template <class U, class ...Args>
    void construct(U *p, Args&& ...args);
    template <class U>
    void destroy(U *p);

    BloombergLP::bslma::Allocator *mechanism() const;





};
```

bsl::allocator<T>

```
template <class T>
class allocator {
    BloombergLP::bslma::Allocator *d_mechanism;

  public:
    typedef T value_type;

    allocator();
    allocator(BloombergLP::bslma::Allocator *mechanism);    // IMPLICIT
    template <class U>
    allocator(const allocator<U>& rhs);

    pointer allocate(size_type n);
    void deallocate(pointer p, size_type n);

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

    BloombergLP::bslma::Allocator *mechanism() const;

};
```

bsl::allocator<T>

```
template <class T>
class allocator {
    BloombergLP::bslma::Allocator *d_mechanism;

  public:
    typedef T value_type;

    allocator();
    allocator(BloombergLP::bslma::Allocator *mechanism);    // IMPLICIT
    template <class U>
    allocator(const allocator<U>& rhs);

    pointer allocate(size_type n);
    void deallocate(pointer p, size_type n);

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

    BloombergLP::bslma::Allocator *mechanism() const;
    allocator select_on_container_copy_construction() const;
};
```

bsl::allocator<T>

```
template <class T>
class allocator {
    BloombergLP::bslma::Allocator *d_mechanism;

  public:
    typedef T value_type;

    allocator();
    allocator(BloombergLP::bslma::Allocator *mechanism);   // IMPLICIT
    template <class U>
    allocator(const allocator<U>& rhs);

    pointer allocate(size_type n);
    void deallocate(pointer p, size_type n);

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

    BloombergLP::bslma::Allocator *mechanism() const;
    allocator select_on_container_copy_construction() const;
};
```

Allocator Propagation

- Allocator is bound at construction
- Should allocator be rebound on assignment?
  - Assignment copies data
  - Allocator is orthogonal, specific to each container object
- Traits give control of the propagation strategy
  - Defaults never propagate

Gotchas for allocators

- ‘smart’ pointers should be iterators, and not manage ownership
- stateful allocators need to share state
  - a copy of an allocator must compare equal, and so be able to deallocate memory supplied by the original

Implementing a container

Example Container

```
template <typename T,
          typename Allocator = allocator<T>>
struct dynarray {
   dynarray(initializer_list<T> data,
            Allocator alloc);
private:
   using AllocTraits = allocator_traits<Allocator>;
   using Pointer     = typename AllocTraits::pointer;

   Pointer    d_data;
   AllocType  d_alloc;
};
```

```
template <typename T, typename Allocator>
void dynarray<T, Allocator>::dynarray(initializer_list<T> data,
                                      Allocator           alloc)
: d_data{}
, d_alloc{alloc}
{
   d_data = AllocTraits::allocate(d_alloc, data.size());
   auto *ptr = addressof(*d_data);
   try {
      for (auto const &elem : data) {
         AllocTraits::construct(d_alloc, ptr, elem);
         ++ptr;
      }
   }
   catch(...) {
      for (auto *base = addressof(*d_data); base != ptr; ++base) {
         AllocTraits::destroy(d_alloc, base);
      }
      AllocTraits::deallocate(d_alloc, d_data, data.size());
      throw;
   }
}
```

```
template <typename T, typename Allocator>
void dynarray<T, Allocator>::dynarray(initializer_list<T> data,
                                      Allocator            alloc)
: d_data{}
, d_alloc{alloc}
{
   d_data = AllocTraits::allocate(d_alloc, data.size());
   auto *ptr = addressof(*d_data);
   try {
      for (auto const &elem : data) {
         AllocTraits::construct(d_alloc, ptr, elem);
         ++ptr;
      }
   }
   catch(...) {
      for (auto *base = addressof(*d_data); base != ptr; ++base) {
         AllocTraits::destroy(d_alloc, base);
      }
      AllocTraits::deallocate(d_alloc, d_data, data.size());
      throw;
   }
}
```

```
template <typename T, typename Allocator>
void dynarray<T, Allocator>::dynarray(initializer_list<T> data,
                                      Allocator           alloc)
: d_data{}
, d_alloc{alloc}
{
   d_data = AllocTraits::allocate(d_alloc, data.size());
   auto ptr = d_data;
   try {
      for (auto const &elem : data) {
         AllocTraits::construct(d_alloc, addressof(*ptr), elem);
         ++ptr;
      }
   }
   catch(...) {
      for (auto base = d_data; base != ptr; ++base) {
         AllocTraits::destroy(d_alloc, addressof(*base));
      }
      AllocTraits::deallocate(d_alloc, d_data, data.size());
      throw;
   }
}
```

```
template <typename T, typename Allocator>
void dynarray<T, Allocator>::dynarray(initializer_list<T> data,
                                      Allocator            alloc)
: d_data{}
, d_alloc{alloc}
{
   d_data = AllocTraits::allocate(d_alloc, data.size());
   auto ptr = d_data;
   try {
      for (auto const &elem : data) {
         AllocTraits::construct(d_alloc, addressof(*ptr), elem);
         ++ptr;
      }
   }
   catch(...) {
      while (d_data != ptr) {
         AllocTraits::destroy(d_alloc, addressof(*--ptr));
      }
      AllocTraits::deallocate(d_alloc, d_data, data.size());
      throw;
   }
}
```

uses_allocator trait

- `template<typename T, typename Alloc>`
  `struct uses_allocator;`
- Derives from `true_type` if:
  - `T` has a nested type alias, `allocator_type`
  - `Alloc` is convertible to `T::allocator_type`
- Otherwise derives from `false_type`
- May be customized for a specific user type, e.g., `tuple`
- Uses-allocator construction passes allocator to element constructor

`scoped_allocator_adapter`

- Allocator adapter to specify allocator or elements in a container
- Can take a pack of allocators, to apply recursively to elements of elements
- Final allocator in pack applies to all deeper nestings
  - typical case is only a single allocator
- e.g. `vector<string, memmap_alloc<string>>`
  - Memory for `vector` in shared memory
  - The `strings` really should be in shared memory too
  - (all using offset-pointers)

Memory Mapped containers

```
namespace memory_mapped {
template <typename T> class mapped_allocator;

template <typename T>
using allocator =
    std::scoped_allocator_adapter<mapped_allocator<T>>;

template <typename T>
using vector = std::vector<T, allocator<T>>;

template <typename T>
using basic_string =
    std::basic_string<T, std::char_traits<T>, allocator<T>>;

using string = basic_string<char>;
}

memory_mapped::vector<memory_mapped::string> vs;
```

Who needs to know how to write `allocator_traits`?

- standard library vendors
-

Who needs to know how to use `allocator_traits`?

- standard library vendors
- container authors

-

Who needs to know how to write an allocator?

- standard library vendors
- container authors(?)
- users with specific requirements

-

Who needs to know how to use an allocator?

- standard library vendors
- container authors
- users with specific requirements
- and everyone else!
-

Bloomberg Allocators

- one possible application of C++11 traits
- allocators derive from `bslma::Allocator`
- allocators are passed by address
- allocator is not part of container type
- per-object allocation
- Next step of standardization: N3525