Allocators in C++11

Who Am I?

- Alisdair Meredith
- Software developer with BloombergLP
- Bloomberg were key sponsors of the allocator work for C++11...
- ... although mostly done before I joined

Motivating Examples

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

Basic Problems

- 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 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: allocators 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 : No returning smart pointers, such as to shared memory

A Brave New World

- C++11 removes 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 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);
  // ...
};
```

```
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;
```

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 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

scoped_allocator_adapter

- `vector<string, memmap_alloc<string>>`
- Memory for `vector` is in shared memory
- The `strings` really should be in shared memory too
  - And using offset-pointers

Example Container

```
template <typename T,
          typename Allocator = std::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;
   }
}
```