C++11 Allocators

Jonathan Wakely

ACCU 2012

- Introduction
- Overview of pre-2011 allocators
- Problems with C++03 allocators
- What's new in 2011
- What changed for users
- What changed for implementors
- Problems with C++11 allocators

What this talk isn't

- Presented by an expert
- A tutorial on writing allocators
- One of the new C++11 features everyone should use or know about

Using Allocators

```
template<typename T,
         typename Alloc = allocator<T> >
  class vector;


vector<int> v;


vector< int, MyAllocator<int> > v1;


MyAllocator<int> a(1024);
vector< int, MyAllocator<int> > v2(a);
```

- Primarily an implementation detail for containers, users rarely need to work with allocators directly

Allocators before 1998

- Allocators were invented by Alex Stepanov as part of the STL
- STL algorithms use iterators to traverse ranges without knowing details of the structure
- STL containers use allocators to manage and manipulate memory without knowing details of the memory model or allocation policy

- Containers need to deal with uninitialized memory and only construct objects in that raw memory on demand

- Containers need a lower-level interface than `new` and `delete`, closer to `malloc()` and `free()`

- Instead of duplicating the logic in every container, the task of creating and destroying objects of type `T` is done by an allocator type such as `allocator<T>`

- Allocators separate memory allocation from construction, and destruction from deallocation

Memory management

```
template <class T>
T* allocate(size_t n, T*);

template <class T>
void deallocate(T* buffer);

template <class T>
Pair<T*, size_t> getTemporaryBuffer(size_t n, T*);

template <class T>
void construct(T* p, const T& value);

template <class T>
void destroy(T* pointer);

template <class T>
void destroy(T* first, T* last);
```


Stepanov, Lee "The Standard Template Library." 1994,
http://www.stepanovpapers.com/Stepanov-The_Standard_Template_Library-1994.pdf

- The STL evolved to classes for memory allocation instead of functions

- This allows customising memory allocation by using different allocator types as a template parameter

```
template<class T, class Alloc = allocator<T> >
  struct vector;
```

- Allocators were meant to encapsulate all the information about the platform's memory model e.g. segmented-memory on early Intel chips could be handled by a custom `Alloc::pointer` type

Allocators in C++03

- The C++ standard states the requirements for allocators (in C++03 see 20.1.5 [lib.allocator.requirements])

- Requirements include nested types that must be defined and expressions that must be valid

  e.g. given an allocator, `a`, of type `A`, the expression `a.allocate(n)` returns an object of type `A::pointer` which refers to uninitialized memory suitable for `n` objects of type `A::value_type`

- The standard also defines the default allocator, `std::allocator`, as a model of that allocator concept (in C++03 see 20.4.1 [lib.default.allocator])
- Looking at the default allocator is a good way to understand the C++03 allocator requirements
- Several nested types in `std::allocator<T>`

```
template<typename 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;
};
```

Aside 1 - Template Typedefs ... Not

- `typedef` declares an alias for an existing type

```
typedef Jonathan Jon;

template<typename T>
  struct not_so_smart_ptr
  {
    typedef std::auto_ptr<T> type;
  };
not_so_smart_ptr<int>::type p(new int);

template<typename T> struct allocator {
  template<typename U> struct rebind {
    typedef allocator<U> other;
  };
};
allocator<int>::rebind<char>::other char_alloc;
```

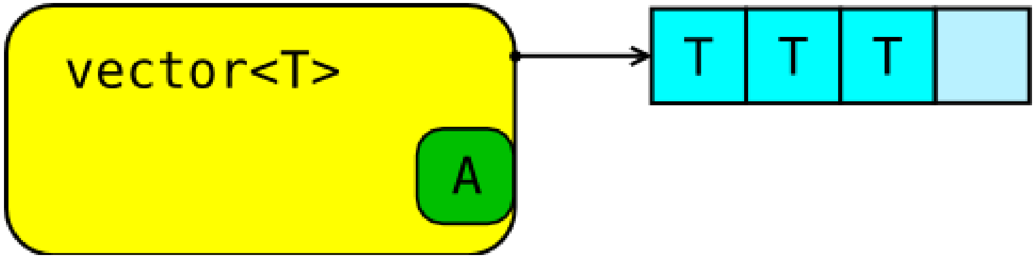
```
allocator();

allocator(const allocator&);

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

~allocator();


pointer allocate(size_type n_objs,
                 allocator<void>::const_pointer hint = 0);

void    deallocate(pointer p, size_type n_objs);


size_type max_size() const;


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

void destroy(pointer p);                    // p->~T()


pointer address(reference r) const;

const_pointer address(const_reference r) const;
```

```
template<>
class allocator<void>
{
public:
  typedef void* pointer;
  typedef const void* const_pointer;
  typedef void value_type;
  template<class U> struct rebind{ typedef allocator<U> other; };
};
```

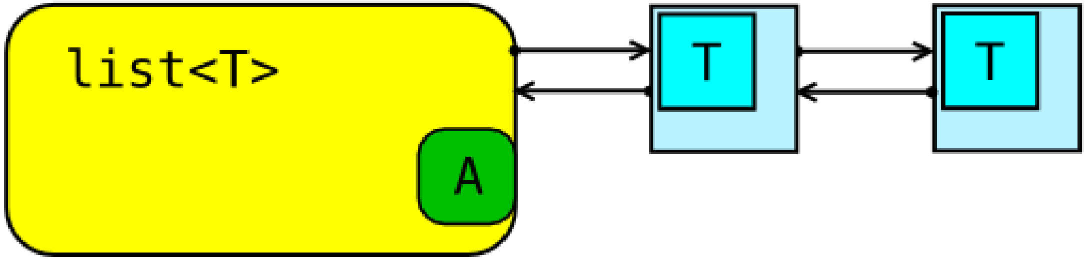
- `allocator<void>::pointer` can be used without instantiating `allocator<void>::reference`

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

- `std::allocator` instances are stateless and so always compare equal
- Allocator equality implies interchangeability i.e. the instances can free each other's memory




```
std::vector<T, A> v(a);

        m_alloc = a;                    // A
        m_data = A::pointer();          // A::pointer
        m_capacity = m_size = 0;        // A::size_type

v.reserve(4);

        m_data = m_alloc.allocate(n);
        m_capacity = n;

v.resize(3, t);

        for(int i = 0; i < n; ++i, ++m_size)
            m_alloc.construct(m_data[m_size], t);
```




```
std::list<T, A> l(a);

      m_alloc = a;

l.push_back(t);

      typedef typename A::template rebind<Node>::other A2;
      A2 a2(m_alloc);
      typename A2::pointer p = a2.allocate(1);
      try {
        a2.construct(p, Node());   // assume no-throw
        m_alloc.construct(&p->data, t);
        m_append_node(p);
      } catch (...) {
        a2.destroy(p, 1);
        a2.deallocate(p, 1);
        throw;
      }
```

What doesn't work well

- Awkward, verbose syntax

```
map<int, double, less<int>, allocator<pair<const int, double> > >
```

- Could have used template template parameters

```
template<class T, template<class> class Alloc> class vector { };
template<class T> class allocator { };
list<int, allocator> l;  // instantiates allocator<int>
                         // and allocator<Node> as needed
```

- As well as the usual sprinkling of `typename`, every `rebind` needs the `template` keyword

```
typedef typename A::template rebind<Node>::other A2;
```

4 Implementations of containers described in this International Standard are permitted to assume that their Allocator template parameter meets the following two additional requirements beyond those in Table 32.

— 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.

5 Implementors are encouraged to supply libraries that can accept allocators that encapsulate more general memory models and that support non-equal instances. In such implementations, any requirements imposed on allocators by containers beyond those requirements that appear in Table 32, and the semantics of containers and algorithms when allocator instances compare non-equal, are implementation-defined.

- The limitations on `pointer` and the other nested types rule out using them to support exotic memory models
- Many interesting use cases for allocators require support for stateful allocators that are not interchangeable

- Might want different allocator instances to allocate from specific pools/arenas to control locality or varying lifetimes

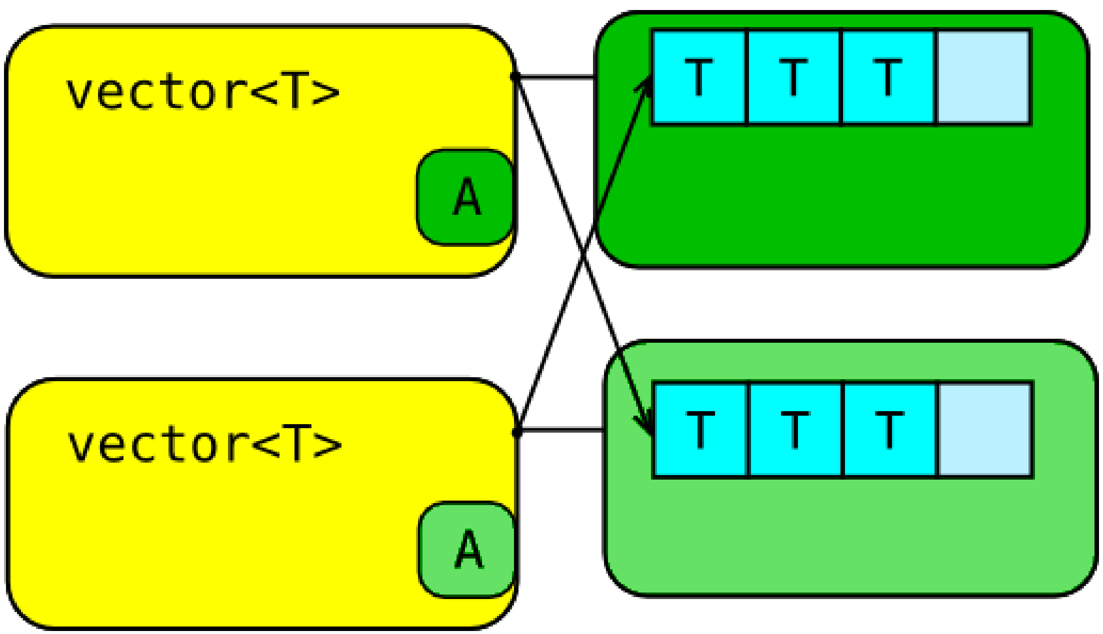
  e.g. a web server might have a pool for global data, another pool for data that persist longer between requests, and short-lived pools used for single requests

- Such allocators need to maintain state and instances using different pools have different identities

- If containers assume all instances of an allocator type are equivalent there's no guarantee that identity will be preserved

- Might want different allocator instances to allocate from specific pools/arenas to control locality or varying lifetimes

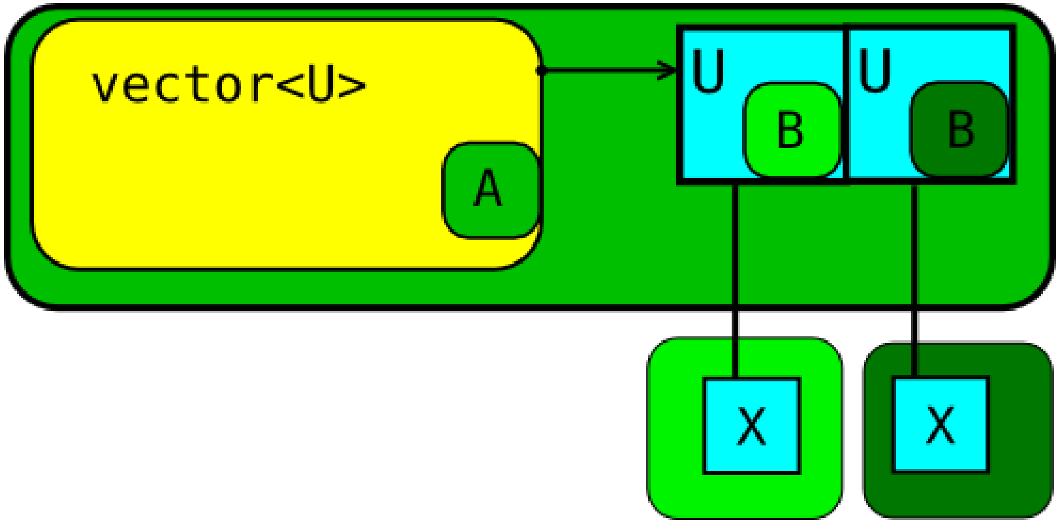
  e.g. a web server might have a pool for global data, another pool for data that persist longer between requests, and short-lived pools used for single requests



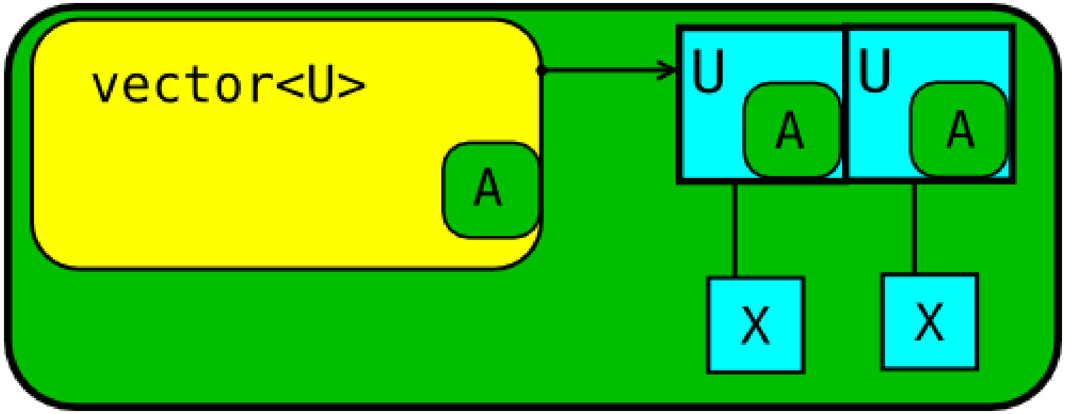
Scoped Allocators

- Consider an allocator that allocates from a region of shared memory accessible from multiple processes
  - It needs to be stateful, to store a handle to the shared memory region it's using
  - It needs to use a custom pointer type that uses an offset into the shared memory region, rather than absolute address
  - (Already deep into implementation-defined territory)

- For the container to be accessible in other processes attached to the same shared memory, the container itself and all its elements *and any memory they use* must come from the same shared memory region
- If the elements use different allocator types, or just unequal instances of the same type, the other process will not be able to use them



- To make this work it's necessary to use the same allocator for the container, its children, its children's children and so on
- This requires that every element can be constructed with an allocator argument which it uses for its own memory allocation and which it then passes to every successive level of object in the structure



- Must be **very** careful to ensure objects use same allocator as the container you're adding them to

```
typedef ShmemAllocator<char> AC;
typedef basic_string<char, char_traits<char>, AC>
  String;
typedef ShmemAllocator<String> AS;
typedef vector<String, AS> Vec;

AC a1(shm_key1);
String s1("hello", a1);
AS a2(shm_key2);
Vec v(a2);

v.push_back("bar");     // v[0] uses AC() !!
v.push_back(s1);        // v[1] uses a1 !!
v.push_back(String(s1.c_str(), v.get_allocator()));
```

What changed in 2011?

- Different allocator requirements
- `std::allocator_traits` provides all the boilerplate
- A container's allocator is not fixed at construction
- The scoped allocator model is not fraught with danger

Using Allocators in 2012

```
template<typename T,
         typename Alloc = allocator<T>>
  class vector;


vector<int> v;


vector<int, MyAllocator<int>> v1;


MyAllocator<int> a{1024};
vector<int, MyAllocator<int>> v2{a};


typedef scoped_allocator_adaptor<MyAllocator<int>>
  SA;
vector<vector<int, MyAllocator<int>>, SA> v3;
```

C++11 Allocator Requirements

- Several new allocator requirements (in C++11 see 17.6.3.5 [allocator.requirements])
- Must be possible to convert `pointer` to `void_pointer` via `static_cast` and vice versa
- `construct()` and `destroy()` take raw pointers not `pointer`
- `construct(T*, Args&&...)` is a variadic function template supporting perfect forwarding
- No longer require `A::reference` and `A::const_reference`

C++11 Allocator Requirements

- New function `select_on_container_copy_construction()`
- Three trait types which must be either `std::true_type` or `std::false_type`
  - `propagate_on_container_copy_assignment`
  - `propagate_on_container_move_assignment`
  - `propagate_on_container_swap`
- I refer to these as POCCA, POCMA and POCS
- These tell containers what to do with their allocators when performing the corresponding operations

Minimal Allocator Interface

```
template<typename T>
  struct Alloc
  {
    Alloc();

    template<typename U>
      Alloc(const Alloc<U>&);

    typename T value_type;

    T*   allocate(std::size_t n);
    void deallocate(T*, std::size_t);
  };

template<typename T>
  bool operator==(const Alloc<T>&, const Alloc<T>&);
template<typename T>
  bool operator!=(const Alloc<T>&, const Alloc<T>&);
```

Pointer Traits

- C++11 adds the `std::pointer_traits` utility to describe properties of pointer-like types

- Used to obtain the pointee type

  e.g. given `T*` obtain `T`

- Transform a type “pointer to A” to type “pointer to B”

  e.g. `shmem_ptr<int>` to `shmem_ptr<const int>`

  `shmem_ptr<int>` to `shmem_ptr<char>`

```
template<class Ptr>
  struct pointer_traits
  {
    typedef Ptr     pointer;
    typedef magic   element_type;
    typedef magic   difference_type;
    template<class U> using rebind = magic;

    static pointer pointer_to(magic);
  };

template<typename T>
  struct pointer_traits<T*>
  {
    typedef T*           pointer;
    typedef T            element_type;
    typedef ptrdiff_t  difference_type;
    template<typename U> using rebind = U*;

    static pointer pointer_to(magic) noexcept;
  };
```

Aside 2 - Alias Declarations

- C++11 adds a new way to declare a `typedef`

```
using Jon = Jonathan;  // typedef Jonathan Jon;

template<typename T>
  using ptr = std::unique_ptr<T>;

template<typename T>
  using smap = std::map<std::string, T>;

smap<int> nums;            // map<string, int>
smap<string> names;        // map<string, string>

template<int N>
  using cbuf = std::array<char, N>;

cbuf<4> currency{ “USD” }; // array<char, 4>
```

- `pointer_traits<Ptr>::element_type` is an alias for
  - `Ptr::element_type` if that type exists, or
  - `T` if `Ptr` is an instantiation like `SomePtr<T,Args...>`

```
template<typename T>
  static true_type  has_et(typename
T::element_type*);
template<typename T>
  static false_type has_et(...);

template<typename>
  struct get_et;
template<template<class T, class... Args> class P>
  struct get_et<P<T, Args...>>
  { typedef T type; };

using element_type = typename
       conditional<decltype(has_et<Ptr>(0))::value,
                   typename Ptr::element_type,
                   typename
```

- `pointer_traits<Ptr>::difference_type` is an alias for
  - `Ptr::difference_type` if that type exists, or
  - `std::ptrdiff_t`
- `pointer_traits<Ptr>::rebind<U>` is an alias for
  - `Ptr::rebind<U>` if that type exists, or
  - `SomePtr<U,Args...>` if `Ptr` is an instantiation like `SomePtr<T,Args...>`
- `pointer_traits::pointer_to(element_type& ref)` returns a pointer to its argument by calling `Ptr::pointer_to(ref)` or `std::addressof(ref)`
  - Not callable when `element_type` is `void`

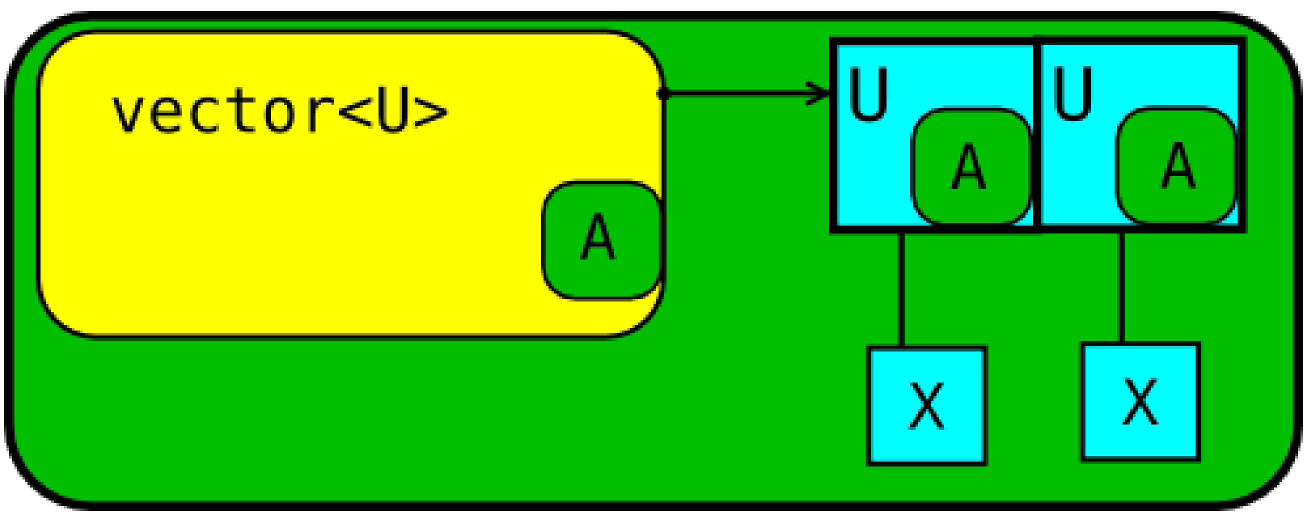
Allocator Traits

- The new class template `std::allocator_traits` provides all the boilerplate that used to be needed by user-defined allocators
- The default definitions provided by the traits class allows C++03 allocators to work unchanged in containers that use the C++11 allocator model
- e.g. containers can use variadic `construct(p, std::forward<Args>(args)...)` to initialize elements even if the allocator only provides `construct(pointer, const T&)`

```
template<typename Alloc>
  struct allocator_traits
  {
    typedef typename Alloc::value_type value_type;

    template<typename T>
      using rebind_alloc = ...;  // Alloc::rebind<T>::other
                                 // or if Alloc is A<X,...>
                                 // then A<T,...>
    template<typename T>
      using rebind_traits
        = allocator_traits<rebind_alloc<T>>;

    typedef ... pointer;        // either Alloc::pointer
                                // or value_type*
    typedef ... const_pointer; // Alloc::const_pointer or
    // pointer_traits<pointer>::rebind<const value_type>

    typedef ... propagate_on_container_copy_assignment;
    typedef ... propagate_on_container_move_assignment;
    typedef ... propagate_on_container_swap;
```

Scoped Allocator Adaptor

- Class template `std::scoped_allocator_adaptor` is provided to adapt existing allocator types so that the parent's allocator will be automatically passed to a child object's constructor if the child type supports it



Scoped Allocator Adaptor

```
typedef ShmemAllocator<int> ShAlloc;
typedef scoped_allocator_adaptor<ShAlloc> ScShA;
vector<vector<int, ShAlloc>, ScShA> v3;
```

- When the container calls `scoped_allocator_adaptor<A>::construct(args)` to construct an element the adaptor detects whether the element type can be constructed with an allocator of type `A` and calls one of:
  - `allocator_traits<A>::construct(std::allocator_arg_t, alloc, args)`
  - `allocator_traits<A>::construct(args, alloc)`
  - `allocator_traits<A>::construct(args)`

- Elements automatically use the same allocator as the container you're adding them to

```
typedef ShmemAllocator<char> AC;
typedef basic_string<char, char_traits<char>, AC>
  String;
typedef ShmemAllocator<String> AS;
typedef scoped_allocator_adaptor<AS> SAS;
typedef vector<String, SAS> Vec;

AC a1(shm_key1);
String s1("hello", a1);
SAS a2(shm_key2);
Vec v(a2);

v.push_back("bar");    // v[0] uses a2
v.push_back(s1);       // v[1] uses a2
```

Scoped Allocator Adaptor

- Another trait, `std::uses_allocator<T, Alloc>`, detects whether the nested type `T::allocator_type` exists and whether that type can be constructed from an object of type `Alloc`
- New "allocator-extended" copy/move constructors are added to many types so they can be passed an allocator when copied/moved
- To say your own type uses-allocator provide appropriate constructors and either define a nested `allocator_type` member or specialize (or partially specialize) `std::uses_allocator` trait

What changed for users

- Easier to define allocators
- Stateful allocators are well-defined and fully supported by containers
- Possible to swap containers with unequal allocators (but might take linear time or throw)
- Safe and fairly simple to use scoped allocators
- Might lose no-throw guarantees on some container operations (depending on quality of implementation)

What changed for implementors

- All containers and several other types gained new “allocator-extended” constructors
- Containers must be updated to do all allocator operations via `allocator_traits`.
- Containers are required to support non-equal allocators (although many did so already)
- Container implementations are significantly more complicated by the propagation traits (POCCA, POCMA and POCS)

```
typedef typename A::template rebind<Node>::other A2;
A2 a2(m_alloc);
typename A2::pointer p = a2.allocate(1);
try {
  a2.construct(p, Node());
  m_alloc.construct(&p->data, t);
  m_append_node(p);
} catch (...) {
  a2.deallocate(p, 1);
  throw;
}
```

```
typedef allocator_traits<A> Atr;
typedef typename ATr::template rebind_traits<Node> ATr2;
typename ATr2::allocator_type a2(m_alloc);
typename ATr2::pointer p = ATr2::allocate(a2, 1);
Node* p2 = nullptr;
try {
  p2 = new (std::addressof(*p)) Node();
  ATr::construct(m_alloc, &p->data, t);
  m_append_node(p);
} catch (...) {
  if (p2) p2->~Node();
  ATr2::deallocate(a2, p, 1);
  throw;
}
```

Allocator propagation

- In C++03 copying a container always copied its allocator.
- If the allocator being copied uses a buffer on the stack or another non-copyable or short-lived source of memory it would be a bad idea to copy the allocator into other containers that will outlive the allocator's source of memory (e.g. by returning an allocator from a function)
- In C++11 a new copy of a container will call `select_on_container_copy_construction()` on the source allocator, instead of just taking a copy

Allocator propagation

- In C++03 a container that is assigned to can reuse its existing allocated memory
- In C++11 if POCCA is true the container being assigned to will have its allocator replaced with a copy of the source container's allocator
- But if the two allocators are not equal then the existing storage must be deallocated before the allocator is replaced, then new storage allocated with the new allocator

- Without allocator propagation, move assigning to a container is simple: the target object takes ownership of the source object's data
- If POCMA is false the target container will *not* have its allocator replaced by the source object's allocator
- If the two allocators are not equal then the target **cannot** take ownership of the source's data and instead must individually move-assign source elements to its own elements, allocating additional memory if required
- This means when POCMA is false, move assignment might take O(n) time and could throw exceptions if re-allocation is needed

- In C++03 swapping containers does not swap the allocators,
- If POCMA is false the target container will *not* have its allocator replaced by the source object's allocator
- If the two allocators are not equal then the target **cannot** take ownership of the source's data and instead must individually move-assign source elements to its own elements, allocating additional memory if required
- This means when POCMA is false, move assignment might take O(n) time and could throw exceptions if re-allocation is needed

Problems with C++11 Allocators

- In order to maintain backwards compatibility, `propagate_on_container_move_assignment` defaults to false, but `std::allocator` does not override the default
- Containers using `std::allocator` (or other allocators with POCMA=false) perform a runtime equality comparison in the move assignment operator, so lose the `noexcept` property on move assignment
- POCMA should be true for `std::allocator`, and authors of stateless allocators should override it

Problems with C++11 Allocators

- No compile-time property to tell if allocators always compare equal
- If `operator==` was allowed to return something "convertible to `bool`" instead of `bool` then equality comparisons for stateless "always equal" allocators could return `std::true_type` instead
- That could be detected at compile-time and used to make container move assignment and swap `noexcept` when using stateless allocators

Problems with C++11 Allocators

- Allocators requirements are incomplete
  - When POCCA is true allocators need to be nothrow CopyAssignable
  - When POCMA is true allocators need to be nothrow MoveAssignable
  - When POCS is true allocators need to be nothrow Swappable

Problems with C++11 Allocators

- `allocator_traits::construct` doesn't use new uniform initialization syntax
- Prevents using "emplace" functions for aggregates:
```
struct S { int i; char c; };
std::vector<S> v;
v.emplace_back(1, 'a');
```
- Changing to always use uniform init would break some code, (`vector{1,3}` is not the same as `vector(1,3)`), but it would be possible to fall-back to uniform init if the other syntax is invalid