How is allocator-aware container assignment implemented?

For example, from **std::deque::operator =** in C++ Reference:
(1) Copy Assignment (const std::deque &other)

> Replaces the contents with a copy of the contents of other.
> If std::allocator_traits::propagate_on_container_copy_assignment() is true, the target allocator is replaced by a copy of the source allocator.
> If the target and the source allocators do not compare equal, the target (*this) allocator is used to deallocate the memory, then other's allocator is used to allocate it before copying the elements.

If `this->get_allocator() == other.get_allocator()`, I can simply destroy and deallocate `this`' elements if needed, or allocate and construct elements if needed, or copy-assign the elements from `other` to `*this` if needed.
But what if not? Does the quote above mean that I can't copy-assign the elements, so I have to destroy and deallocate ALL the elements first, using `this->get_allocator()`, and then allocate and construct the elements, using `other.get_allocator()`?
But if that is the case, why should I use `other.get_allocator()` for the allocation?
Won't it cause some runtime error later, as `this` won't deallocate the memory properly?

(2) Move Assignment (std::deque &&other)

> Replaces the contents with those of other using move semantics (i.e. the data in other is moved from other into this container). other is in a valid but unspecified state afterward. If std::allocator_traits::propagate_on_container_move_assignment() is true, the target allocator is replaced by a copy of the source allocator. If it is false and the source and the target allocators do not compare equal, the target cannot take ownership of the source memory and must move-assign each element individually, allocating additional memory using its own allocator as needed. In any case, all element originally present in *this are either destroyed or replaced by elementwise move-assignment.

If `this->get_allocator() == other.get_allocator()`, this is an easy task.
But if not, the same questions above follow, except in this case move-assignment is used.

In both cases, I have an additional question.
If the elements can neither be copy-assigned or move-assigned, is it okay to destroy it and construct from other? If it is, whose allocator should I use?

c++ c++11 memory-management assign allocator

edited Jan 17 at 7:12

asked Nov 25 '16 at 9:34
Dannyu NDos
495 4 14

2 Answers

A POCCA (propagate-on-container-copy-assignment) allocator is copy-assigned as part of the container's copy assignment. Likewise, a POCMA allocator is move-assigned when the container's move assigned.

> Does the quote above mean that I can't copy-assign the elements, so I have to destroy and deallocate ALL the elements first, using `this->get_allocator()`, and then allocate and construct the elements, using `other.get_allocator()`?

Correct.

> But if that is the case, why should I use `other.get_allocator` for the allocation? Won't it cause some runtime error later, as `this->get_allocator()` won't deallocate the memory properly?

Because the assignment propagates the allocator: after the assignment, `this->get_allocator()` is a copy of `other.get_allocator()`, so it can safely deallocate memory allocated by it.

> If `this->get_allocator() == other.get_allocator()`, this is an easy task. But if not, the same questions above follow, except in this case move-assignment is used.

Actually, this is completely different. Move assignment with a POCMA allocator is trivial: you destroy all the elements in `*this`, free the memory, and plunder the memory and allocator of `other`.

The only case where container move assignment has to resort to element-wise move assignment/construction is when you have a *non-POCMA* allocator and the allocators compare unequal. In that case, all allocation and construction are done with `this->get_allocator()` since you don't propagate anything.

> In both cases, I have an additional question. If the elements can neither be copy-assigned or move-assigned, is it okay to destroy it and construct from other? If it is, whose allocator should I use?

Destroy it using the allocator it was originally constructed with; construct it using the allocator it will be destroyed with. In other words, if you are propagating the allocator, then destroy it with the target allocator and construct with the source allocator.

answered Nov 25 '16 at 10:12

T.C.
87.4k 11 166 261

What about non-POCCA allocators? Do I have to use `this->get_allocator()` any time? – Dannyu NDos Nov 25 '16 at 10:22

1 Well, yes, if you need to allocate. – T.C. Nov 25 '16 at 10:28

I am answering my own question to show what I got. --Dannyu NDos, 2017 Jan 16

Either in the copy or move assignment, its behavior depends on two conditions:
1. is the allocators compare equal? (That is, is the source allocator able to destroy and deallocate the target container's elements?)
2. does the source's allocator propagate (= be assigned to target) during container assignment?

For copy assignment:
A. If the allocators compare equal:
Directly copy-assigning elements to elements can be safely done.
As the allocators already compare equal, it doesn't matter whether the allocator propagates. If any element needs to be constructed or destroyed, it also doesn't matter whose allocator does it.
B. If the allocators don't compare equal:
B.a. If the allocator doesn't propagate:
Directly copy-assigning elements to elements can be safely done, but if any element needs to be constructed or destroyed, the source allocator must do it, as only it can destroy target container's elements.
B.b. If the allocator propagates:
First, the target allocator must destory and deallocate all the target container's elements.
And then the allocator propagates, and then the source allocator allocates and copy-constructs all the source container's elements.

For move assignment:
A. If the allocators compare equal:
The target container erases all its elements, and then takes ownership of the source container's elements. This takes O(1) time.
B. If the allocators don't compare equal:
B.a. If the allocator doesn't propagate:
Directly move-assigning elements to elements can be safely done, but if any element needs to be constructed or destroyed, the source allocator must do it, as only it can destroy source container's element. This takes O(n) time. The source container must be in valid state after assignment.
B.b. If the allocator propagates:
First, the target allocator must destory and deallocate all the target container's elements.
And then the allocator propagates, and then the source allocator allocates and move-constructs all the source container's elements. This takes O(n) time. The source container must be in valid state after assignment.

In source code, given `alloc` is container's allocator, `Alloc` is its type, they are generally written like this:

```
/*container*/ &operator = (const /*container*/ &other) {
    if (std::allocator_traits<Alloc>::propagate_on_container_copy_assignment::value
&& alloc != other.alloc) {
        clear();
        alloc = other.alloc;
        // directly copy-constructs the elements.
    } else {
        // directly copy-assigns the elements.
        // alloc does all allocation, construction, destruction, and deallocation
as needed.
```

```
    }
    return *this;
}
/*container*/ &operator = (/*container*/ &&other)
noexcept(std::allocator_traits<Alloc>::is_always_equal::value) {
    if (alloc == other.alloc) {
        clear();
        // *this takes ownership of other's elements.
    } else if
(std::allocator_traits<Alloc>::propagate_on_container_move_assignment::value) {
        clear();
        alloc = other.alloc;
        // directly move-constructs the elements.
    } else {
        // directly move-assigns the elements.
        // alloc does all allocation, construction, destruction, and deallocation
as needed.
    }
    // the source container is made valid, if needed.
    return *this;
}
```

answered Jan 16 at 4:31

Dannyu NDos

495 4 14