Howard Hinnant
2016-08-23

Allocator Boilerplate

Contents

- Introduction
- C++11 and forward
- C++03 and backward

Introduction

This paper does not introduce any novel ideas. Nor is it a complete tutorial on allocators. The only purpose of this paper is to be a quick go-to site for copy/pasting the code you need to create your custom allocator. There is nothing at all novel in the code presented here. Just copy/paste the allocator skeleton here and insert *your own* novel code! Do not worry about copyright issues on copy/pasting code out of this document. There is no code here that is worthy of copyright. It is *all* just boilerplate. The hope is that by lowering the barrier just a little bit, more people can more easily create truly useful custom allocators.

In case it is helpful, two allocator skeletons are presented:

- C++11 and forward
- C++03 and backward

If you are using a compiler/library that has only a partial implementation for C++11 allocator support, you can easily copy/paste what you need from both of these skeletons thus creating a custom hybrid C++03/C++11 solution.

C++11 and forward

Below much of the skeleton is commented out. The commented out code represents functionality that `std::allocator_traits<allocator<T>>` defaults for you, if you do not provide it. The implementation in the code shows you exactly what the defaults are. Thus if you simply uncomment the code, you will not get any functionality changes. To get something different from what the defaults provide, uncomment and change the implementation.

```
template <class T>
class allocator
{
public:
    using value_type    = T;

//     using pointer       = value_type*;
//     using const_pointer = typename std::pointer_traits<pointer>::template
//                                                     rebind<value_type const>;
//     using void_pointer       = typename std::pointer_traits<pointer>::template
//                                                           rebind<void>;
//     using const_void_pointer = typename std::pointer_traits<pointer>::template
//                                                           rebind<const void>;

//     using difference_type = typename std::pointer_traits<pointer>::difference_type;
//     using size_type       = std::make_unsigned_t<difference_type>;

//     template <class U> struct rebind {typedef allocator<U> other;};

    allocator() noexcept {}  // not required, unless used
    template <class U> allocator(allocator<U> const&) noexcept {}
```

```
    value_type*  // Use pointer if pointer is not a value_type*
    allocate(std::size_t n)
    {
        return static_cast<value_type*>(::operator new (n*sizeof(value_type)));
    }

    void
    deallocate(value_type* p, std::size_t) noexcept  // Use pointer if pointer is not a value_type*
    {
        ::operator delete(p);
    }

//     value_type*
//     allocate(std::size_t n, const_void_pointer)
//     {
//         return allocate(n);
//     }

//     template <class U, class ...Args>
//     void
//     construct(U* p, Args&& ...args)
//     {
//         ::new(p) U(std::forward<Args>(args)...);
//     }

//     template <class U>
//     void
//     destroy(U* p) noexcept
//     {
//         p->~U();
//     }

//     std::size_t
//     max_size() const noexcept
//     {
//         return std::numeric_limits<size_type>::max();
//     }

//     allocator
//     select_on_container_copy_construction() const
//     {
//         return *this;
//     }

//     using propagate_on_container_copy_assignment = std::false_type;
//     using propagate_on_container_move_assignment = std::false_type;
//     using propagate_on_container_swap            = std::false_type;
//     using is_always_equal                        = std::is_empty<allocator>;
};

template <class T, class U>
bool
operator==(allocator<T> const&, allocator<U> const&) noexcept
{
    return true;
}

template <class T, class U>
bool
operator!=(allocator<T> const& x, allocator<U> const& y) noexcept
{
    return !(x == y);
}
```

So copy the above into your code. Change the name from `allocator` to whatever makes sense for you. Delete all of the comments for which the defaults provided by `std::allocator_traits` meet your needs. For whatever is left, fill in your implementation.

Notes:

- `is_always_equal` is new for C++1y (hopefully that will be C++17). As I write this paper, it is not likely to actually be used, is not standard, and is subject to change.

- The default implementation for `max_size()` is not incredibly useful. I have submitted an LWG issue to change the default to:

  ```
  return std::numeric_limits<size_type>::max() / sizeof(value_type);
  ```

  which makes a lot more sense when `sizeof(value_type) > 1`.

- Under discussion is the possibility to remove the requirement that you provide `operator==` and `operator!=` if `is_always_equal{}` is `true`.

- The nested types `reference` and `const_reference` are no longer required in C++11 (as they were in C++03).

- The member functions `address(reference)` and `address(const_reference)` are no longer required in C++11 (as they were in C++03).

- Your allocator must be `CopyConstructible` and `MoveConstructible`. If `propagate_on_container_copy_assignment{}` is true, your allocator must be `CopyAssignable`. If `propagate_on_container_move_assignment{}` is true, your allocator must be `MoveAssignable`. If `propagate_on_container_swap{}` is true, your allocator must be `Swappable`. If they exist, these operations should not propagate an exception out. However they do not need to be marked with `noexcept`. However I recommend marking them with `noexcept` if the compiler does not implicitly do so, so that traits such as `is_nothrow_copy_constructible<allocator<T>>` give the right answer.

- If two allocators compare equal, that means that they can deallocate each other's allocated pointers. If two instances of your allocators can't do this, they must **not** compare equal to each other, else run time errors will result. However copies, even converting copies, *are required* to compare equal.

C++03 and backward

Here is the C++98/C++03 allocator skeleton. In this case, there is no such thing as `std::allocator_traits` and so nothing is defaulted for you. You will know if you need anything from this as you will get compile-time errors if your implementation is asking for it, and you don't have it.

```
template <class T> class allocator;

template <>
class allocator<void>
{
public:
    typedef void              value_type;
    typedef value_type*       pointer;
    typedef value_type const* const_pointer;
    typedef std::size_t       size_type;
    typedef std::ptrdiff_t    difference_type;

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

template <class T>
class allocator
{
public:
    typedef T                 value_type;
    typedef value_type&       reference;
    typedef value_type const& const_reference;
    typedef value_type*       pointer;
    typedef value_type const* const_pointer;
    typedef std::size_t       size_type;
```

```
        typedef std::ptrdiff_t    difference_type;

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

        allocator() throw() {}  // not required, unless used
        template <class U> allocator(allocator<U> const& u) throw() {}

        pointer
        allocate(size_type n, allocator<void>::const_pointer = 0)
        {
            return static_cast<pointer>(::operator new (n*sizeof(value_type)));
        }

        void
        deallocate(pointer p, size_type)
        {
            ::operator delete(p);
        }

        void
        construct(pointer p, value_type const& val)
        {
            ::new(p) value_type(val);
        }

        void
        destroy(pointer p)
        {
            p->~value_type();
        }

        size_type
        max_size() const throw()
        {
            return std::numeric_limits<size_type>::max() / sizeof(value_type);
        }

        pointer
        address(reference x) const
        {
            return &x;
        }

        const_pointer
        address(const_reference x) const
        {
            return &x;
        }
    };

    template <class T, class U>
    bool
    operator==(allocator<T> const&, allocator<U> const&)
    {
        return true;
    }

    template <class T, class U>
    bool
    operator!=(allocator<T> const& x, allocator<U> const& y)
    {
        return !(x == y);
    }
```

As is evident, there is a lot more boilerplate required in C++98/03 than in C++11. Also C++98/03 allocators do not portably support pointer types that are not `value_type*`. And support for allocators that do not compare equal is not portable.