IX. Allocators and basic memory management

Let's look at vector's template parameters list

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

Every std container has an allocator. Why?

Let's say we have a list, and we **know**, that there will be **lots** of insertions. That means, that there will be many *new* calls, which will be time consuming (even though it is still O(1))

Take another example. Let's say we already have a batch of memory (*on stack*), that we want to use **instead** of heap (For example, when we don't need much memory). That would be quiet a bit more efficient than calling new.

That's where allocators come in!

Our scheme was as follows:

- Container –> operator new –> malloc –> os

Now it is:

- Container –> Allocator –> operator new –> malloc –> os

https://en.cppreference.com/w/cpp/named_req/Allocator

The main functions of allocator are:

- Allocate
- Deallocate
- Destroy (now optional)
- Construct (now optional)

Old Vector review

Let's then modify our Vector class from the last semester:

```
template <typename T, typename Alloc = std::allocator<T>>
class Vector {
    T* arr = nullptr;
    size_t sz = 0;
    size_t cap = 0;
    [[no unique address]] Alloc alloc; // EBO
  public:
...
  Vector(size_t n, const T& value)
    : sz(n), cap(n)
  {
    arr = alloc.allocate
    // arr = reinterpret_cast<T*>(new char[n * sizeof(T)]);
    size_t i = 0;
    try {
        for (; i < n; ++i) {
            alloc.construct(arr + i, value);
            // new (arr + i) T(value);
        }
    } catch (...) {
        for (size_t j = 0; j < i; ++j) {
            alloc.destroy(arr + i);
            // (arr + i)->~T();
        }
        alloc.deallocate(arr, n);
        // delete[] ...
        throw;
    }
  }
}
```

Note: we could instead do this:

```
template <typename T, template <typename> typename Alloc = std::allocator>
class Vector {
    ...
}
```

However we don't know how many (and which) template arguments Alloc type requires.

Note: below C++20, we used to inherite vector from *private Alloc* to use EBO (empty base optimization), so that Alloc will be free to hold (in memory)

Note: since C++20, you just write [[no unique address]] to **do literally the same** without ugly inheritance.

Basic allocator implementation for our vector

```
template <typename T>
struct allocator {
    using value_type = T;

    T* allocate(size_t count) {
        // alternatively:
        // retun ::operator_new(count * sizeof(T));
        return reinterpret_cast<T*>(new std::byte[n * sizeof(T)]);
    }

    void deallocate(T* ptr, size_t) {
        delete[] reinterpret_cast<std::byte*>(ptr);
    }

    template <typename U, typename... Args>
    void construct(U* ptr, const Args&... args) {
        new (ptr) U(args...);
    }

    void destroy(T* ptr) {
        ptr->~T();
    }
};
```

Allocator for list

Here we face *a small problem*. Here we allocate space for **Nodes** instead of values of type T. However out allocator Allocates for type T. How can the user give allocator to container, if he doesn't know the type of Node?

Let's look at our list class from the last lecture with allocator:

```
template <typename T, typename Alloc = std::allocator<T>>
class List {
    struct BaseNode {
        BaseNode* next;
        BaseNode* prev;
    };

    struct Node : BaseNode {
        T value;
    };

    BaseNode fakeNode;
    size_t sz;

    // The word typename is optional since C++20
    using NodeAlloc = typename Alloc::template rebind<Node>::other;

    [[no_unique_address]] NodeAlloc nodeAlloc;
public:

};
```

So, we need some type "NodeAlloc" that creates nodes. How do we get it?

Here is the solution:

```
struct allocator {
    ...
    ...
    ...

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

struct List {
    ...
    BaseNode fakeNode;
    size_t sz;

    // The word typename is optional since C++20
    using NodeAlloc = typename Alloc::template rebind<Node>::other;

    [[no_unique_address]] NodeAlloc nodeAlloc;
}
```

So we just get the version of allocator, **that we actually** need. (Yes, another crutch)

Allocator strategies

std::allocator

Literally just a proxy for operator new

Allocator of Mescherin-Kruchevski-Torvalds (PoolAllocator)

Just allocate a lot of memory (on heap), and deallocate nothing!

Stack allocator

Allocate a chunk of memory on stack and use it (again, deallocate does nothing)

Free List Allocator

We allocate as usual, **but** when we are asked to **deallocate** a block of memory, we remember it, and, when we are asked to allocate a block of memory of the same size, we recall that block and return it instead.

boost::aligned_alloc

Behaves like standard allocator, but operates with alignment N.

Review of functionality

The key functions are *allocate* and *deallocate* (*destroy* and *construct* are mostly the same among almost all allocators, so they became optional soon).

Exception: scoped-allocator

Destroy and construct functions are **so simple**, that their standard implementation was moved to allocat_traits.

In fact, every std container uses allocatetors via allocator_traits — the uniform interface for allocators.

```
template <typename Alloc>
struct alloc_traits {
    template <typename... U, typenmame... Args>
    static void construct(Alloc& alloc, U* ptr, const Args&... args) {
        // it is actually Args&&... args, but we don't know move yet
        if constexpr (/*Alloc has method construct*/) {
            alloc.construct(ptr, args...);
        } else {
            new (ptr) U(args...);
        }
    }

    static void destroy(T* ptr) {
        ptr->~T();
    }
};
```

The question is how to implement that comment? In C++20 it is made with concepts, in earlier versions - not so easy, but doable.

Let's take another take at vector, now with alloc_traits in mind:

```
  Vector(size_t n, const T& value)
    : sz(n), cap(n)
  {
    arr = allocator_traits<alloc>::allocate(n).
    // arr = reinterpret_cast<T*>(new char[n * sizeof(T)]);
    size_t i = 0;
    try {
        for (; i < n; ++i) {
            allocator_traits<alloc>::construct(arr + i, value);
            // new (arr + i) T(value);
        }
    } catch (...) {
        for (size_t j = 0; j < i; ++j) {
            allocator_traits<alloc>::destroy(arr + i);
            // (arr + i)->~T();
        }
        allocator_traits<alloc>::deallocate(arr, n);
        // delete[] ...
        throw;
    }
}
```

Unsolved problems

We will need to think about:

- If the allocator is not empty (in memory), we don't know, whether it should be copied between containers. (The truth is that it depends, more on that link on cppreference)