19.2. A User-Defined Allocator

Allocators provide an interface to allocate, create, destroy, and deallocate objects (Table 19.1). With allocators, containers and algorithms can be parametrized based on the way the elements are stored. For example, you could implement allocators that use shared memory or that map the elements to a persistent database.

Table 19.1. Fundamental Allocator Operations

Expression	Effect
`a.allocate(num)`	Allocates memory for `num` elements
`a.construct(p,val)`	Initializes the element to which `p` refers with `val`
`a.destroy(p)`	Destroys the element to which `p` refers
`a.deallocate(p,num)`	Deallocates memory for `num` elements to which `p` refers

Writing your own allocator is not very hard. The most important issue is how you allocate or deallocate the storage. For the rest, appropriate defaults are usually provided since C++11. (Before C++11, you had to implement the rest in a pretty obvious way.) As an example, let's look at an allocator that behaves just like the default allocator:

Click here to view code image

```
// alloc/myalloc11.hpp

#include <cstddef>    // for size_t

template <typename T>
class MyAlloc {
  public:
    // type definitions
    typedef T value_type;

    // constructors
    // - nothing to do because the allocator has no state
    MyAlloc () noexcept {
    }
    template <typename U>
    MyAlloc (const MyAlloc<U>&) noexcept {
        // no state to copy
    }

    // allocate but don't initialize num elements of type T
    T* allocate (std::size_t num) {
        // allocate memory with global new
        return static_cast<T*>(::operator new(num*sizeof(T)));
    }

    // deallocate storage p of deleted elements
    void deallocate (T* p, std::size_t num) {
        // deallocate memory with global delete
        ::operator delete(p);
    }
};

// return that all specializations of this allocator are interchangeable
template <typename T1, typename T2>
bool operator== (const MyAlloc<T1>&,
                 const MyAlloc<T2>&) noexcept {
    return true;
}
template <typename T1, typename T2>
bool operator!= (const MyAlloc<T1>&,
                 const MyAlloc<T2>&) noexcept {
    return false;
}
```

As the example demonstrates, you have to provide the following features:

- A type definition of the `value_type`, which is nothing but the passed template parameter type.
- A constructor.

- A template constructor, which copies the internal state while changing the type. Note that a template constructor does not suppress the implicit declaration of the copy constructor (see Section 3.2, page 36).
- A member `allocate()`, which provides new memory.
- A member `deallocate()`, which releases memory that is no longer needed.
- Constructors and a destructor, if necessary, to initialize, copy, and clean up the internal state.
- Operators `==` and `!=`.

You don't have to provide `construct()` or `destroy()`, because their default implementations usually work fine (using *placement new* to initialize the memory and calling the destructor explicitly to clean it up).

Using this base implementation, you should find it easy to implement your own allocator. You can use the core functions `allocate()` and `deallocate()` to implement your own policy of memory allocation, such as reusing memory instead of freeing it immediately, using shared memory, mapping the memory to a segment of an object-oriented database, or just debugging memory allocations. In addition, you might provide corresponding constructors and a destructor to provide and release what `allocate()` and `deallocate()` need to fulfill their task.

Note that before C++11, you had to provide a lot more members, which, however, were easy to provide. See *alloc/myalloc03.hpp* for a corresponding complete example, which is also covered in the supplementary section about allocator details at http://www.cppstdlib.com.