19.3. Using Allocators as a Library Programmer

This section describes the use of allocators from the viewpoint of people who use allocators to implement containers and other components that are able to handle different allocators. This section is based, with permission, partly on Section 19.4 of Bjarne Stroustrup's *The C++ Programming Language*, 3rd edition (see [*Stroustrup:C++*]).

As an example, let's look at a naive implementation of a vector. A vector gets its allocator as a template or a constructor argument and stores it somewhere internally:

Click here to view code image

```
namespace std {
    template <typename T,
              typename Allocator = allocator<T> >
    class vector {
        ...
      private:
        Allocator alloc;        // allocator
        T*        elems;        // array of elements
        size_type numElems;     // number of elements
        size_type sizeElems;    // size of memory for the elements
        ...
      public:
        // constructors
        explicit vector(const Allocator& = Allocator());
        explicit vector(size_type num, const T& val = T(),
                        const Allocator& = Allocator());
        template <typename InputIterator>
        vector(InputIterator beg, InputIterator end,
               const Allocator& = Allocator());
        vector(const vector<T,Allocator>& v);
        ...
    };
}
```

Note that strictly speaking the type of `elems` has to be as follows, since C++11:

```
allocator_traits<Allocator>::pointer elems;
```

As for iterator traits (see Section 9.5, page 467), *allocator traits* were introduced to serve as common interface for generic code dealing with allocators. They `provide` types such as pointer and operations such as `allocate()`, `construct()`, `destroy()`, and `deallocate()`.

The second constructor that initializes the vector by `num` elements of value `val` could be implemented as follows:

Click here to view code image

```
namespace std {
    template <typename T, typename Allocator>
    vector<T,Allocator>::vector(size_type num, const T& val,
                                const Allocator& a)
     : alloc(a)     // initialize allocator
    {
        // allocate memory
        sizeElems = numElems = num;
        elems = allocator_traits<Allocator>::allocate(alloc, num);

        // initialize elements
        for (size_type i=0; i<num; ++i) {
            // initialize ith element
            allocator_traits<Allocator>::construct(alloc, &elems[i],
val);
        }
    }
}
```

Note that this code is still not complete because the handling of exceptions is missing. In a proper implementation, if the construction of any element fails, all the allocated memory must be freed.

For the initialization of uninitialized memory, the C++ standard library provides some convenience functions (Table 19.2). Before C++11, the implementation of the constructor was simpler using these functions and not having allocator traits:

```
namespace std {
    template <typename T, typename Allocator>
    vector<T,Allocator>::vector(size_type num, const T& val,
                                const Allocator& a)
     : alloc(a)     // initialize allocator
    {
        // allocate memory
        sizeElems = numElems = num;
        elems = alloc.allocate(num);

        // initialize elements
        uninitialized_fill_n(elems, num, val);
    }
}
```

Table 19.2. Convenience Functions for Uninitialized Memory

Expression	Effect
`uninitialized_fill(`*beg*`,`*end*`,`*val*`)`	Initializes [*beg*,*end*) with *val*
`uninitialized_fill_n(`*beg*`,`*num*`,`*val*`)`	Initializes *num* elements starting from *beg* with *val*
`uninitialized_copy(`*beg*`,`*end*`,`*mem*`)`	Initialize the elements starting from *mem* with the elements of [*beg*,*end*)
`uninitialized_copy_n(`*beg*`,`*num*`,`*mem*`)`	Initialize *num* elements starting from *mem* with the elements starting from *beg* (since C++11)

Since C++11, however, `uninitialized_fill_n()` and `uninitialized_copy()` cannot be used, because they do not use the allocator traits to construct the elements. Without allocator traits, memory management might be broken by user-defined code that specializes allocator properties and/or allocator types such that the actual call of `construct()` may do additional or perhaps completely different operations.

The member function `reserve()` , which reserves more memory without changing the number of elements (see Section 7.3.1, page 271), could be implemented as follows:

```
namespace std {
    template <typename T, typename Allocator>
    void vector<T,Allocator>::reserve (size_type size)
    {
        // reserve() never shrinks the memory
        if (size <= sizeElems) {
            return;
        }

        // allocate new memory for size elements
        T* newmem = allocator_traits<Allocator>::allocate(alloc, size);

        // copy old elements into new memory
        ...

        // destroy old elements
        for (size_type i=0; i<numElems; ++i) {
            allocator_traits<Allocator>::destroy(alloc, &elems[i]);
        }

        // deallocate old memory
        allocator_traits<Allocator>::deallocate(alloc, elems,
sizeElems);

        // so, now we have our elements in the new memory
        sizeElems = size;
        elems = newmem;
    }
}
```

Again, this code is over-simplified: The tricky part, copying the elements into the new memory, is missing because this code has to deal with exceptions and should call move operations instead of copy operations if possible.

Raw Storage Iterators

In addition, class `raw_storage_iterator` is provided to iterate over uninitialized memory to initialize it. Therefore, you can use any algorithms with a `raw_storage_iterator` to initialize memory with the values that are the result of that algorithm.

For example, the following statement initializes the storage to which `elems` refers by the values in range [

`x.begin()` , `x.end()`) :

```
copy (x.begin(), x.end(),                        // source
      raw_storage_iterator<T*,T>(elems));    // destination
```

The first template argument (`T*` , here) has to be an output iterator for the type of the elements. The second template argument (`T` , here) has to be the type of the elements.

Temporary Buffers

In code, you might also find the `get_temporary_buffer()` and `return_temporary_buffer()` . They are provided to handle uninitialized memory that is provided for short, temporary use inside a function. Note that `get_temporary_buffer()` might return less memory than expected. Therefore, `get_temporary_buffer()` returns a pair containing the address of the memory and the size of the memory (in element units). Here is an example of how to use it:

```
void f()
{
    // allocate memory for num elements of type MyType
    pair<MyType*,std::ptrdiff_t> p
     = get_temporary_buffer<MyType>(num);
    if (p.second == 0) {
        // could not allocate any memory for elements
        ...
    }
    else if (p.second < num) {
        // could not allocate enough memory for num elements
        // however, don't forget to deallocate it
        ...
    }

    // do your processing
    ...

    // free temporarily allocated memory, if any
    if (p.first != 0) {
        return_temporary_buffer(p.first);
    }
}
```

However, it is rather complicated to write exception-safe code with `get_temporary_buffer()` and `return_temporary_buffer()` , so they are usually no longer used in library implementations.