Doc No: X3J16/96-0181
WG21/N0999
Date: September 24, 1996
Project: Programming Language C++
Reply to: Bjarne Stroustrup

Allocators

As readers of -lib have noticed, I'm unhappy with the definition of allocators in the current WP. I'm worried enough to want a serious discussion with the aim of either getting a better definition or a much better understanding of why what is there is the best we can give people.

My conjecture is that the current allocator definition doesn't serve the needs for which we introduced allocators, is too complicated to use, and that a simpler design will do what we intented - in a simpler and more efficient way.

In other words:

I think allocators are broken, and that I know a way to fix them.

I think allocators address (or can address) three concerns:

(1) alternative allocators: That is, by supplying an non-standard allocator a programmer can escape from the implementation supplied default and provide a special-purpose allocator that supposedly will serve some container implementors (and their users) better.

Unless you really trust your implementor(s) always to do better than you can, this an important facility. My experience with vendor-supplied general purpose allocators have been distinctly mixed.

(2) alternative pointers: That is, by non-standard allocator a programmer (or an implementor) can supply more than one pointer type. The PC memory models is an example, so would a more modern architecture with 32 and 72 bit pointers be.

I am not convinced we have seen the last architecture with more than one pointer type.

(3) alternative accessors: That is, allowing a user to design pointer (and reference) types that allows controlled access to data holding more than just a pointer.

I don't think C++ can support a totally transparent and perfectly general scheme (the requirement that a pointer type can be converted to a void* gives the user a loop-hole). However, less general and more application-specific accessors (such as range-checked pointers) are valuable. I consider accessors especially useful for experimentation.

Let me discuss some alternatives for allocators:

(1) ''No allocators''

Pro:

It couldn't be simpler.

Con:

Requires sweeping changes to the working paper.

```
        No alternative allocators.
        No alternative pointers.
        No alternative accessors.
        We don't have any typedef to play with (ptrdiff_t and size_t is it).
My rating:
        Unacceptable.


(2) ''Status quo''

template <class T> class allocator {
public:
        typedef T value_type;
        typedef size_t size_type;
        typedef ptrdiff_t difference_type;

        typedef T* pointer;
        typedef const T* const_pointer;

        typedef T& reference;
        typedef const T& const_reference;

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

        allocator() throw();
        template <class U> allocator(const allocator<U>&) throw();
        template <class U> allocator& operator= (const allocator<U>&) throw();
        ~allocator() throw();

        pointer address(reference r) const;
        const_pointer address(const_reference r) const;

        pointer allocate(size_type, typename allocator<void>::const_pointer hint = 0);
        void deallocate(pointer p, size_type n);
        size_type max_size() const throw();

        void construct(pointer p, const T& val);
        void destroy(pointer p);
};

Pro:
        It's status quo.
        The working group recommended it and we voted it in.
        Alternative allocators.
Con:
        Its use is very obscure/difficult_to_use/error-prone (have a
                look at rebind).
        Limited alternative pointers (only things that can be converted
                to/from void* can be used to access objects).
        No alternative accessors.
My rating:
        Not good enough. Too complicated for what it does. Too hard
        to understand, use, and teach.


(3) ''Pre-Stockholm status quo''

class allocator {
public:
        typedef size_t size_type;
        typedef ptrdiff_t difference_type;

        template<class T> struct types {
                typedef T*        pointer;
                typedef const T*  const_pointer;
                typedef T&        reference;
```

```
                typedef const T&  const_reference;
                typedef T         value_type;
        };

        allocator();
        ~allocator();

        template<class T>
                typename types<T>::pointer
                        address(types<T>::reference) const;
        template<class T>
                typename types<T>::const_pointer
                        address(types<T>::const_reference) const;

        template<class T, class U>
                typename types<T>::pointer
                        allocate(size_type, types<U>::const_pointer hint = 0);
        template<class T> void deallocate(types<T>::pointer);
        template<class T> size_type max_size() const;

        template <class T1, class T2> void construct(T1* p, const T2& val);
        template <class T> void destroy(T* p);
};
```

Pro:

Alternative allocators.
Alternative pointers.
Alternative accessors.

Con:

The working group replaced it; it must have deficiencies that I have overlooked. The "types" structure and the allocate() that relies on it is somewhat obscure. A class of allocate for an "other" type requires the use of one of the most obscure notations in the language (the "template" qualification of an explicitly qualified member template):

```
                allocator.template allocate<link>(1);
```

My rating:

I thought it was good enough (but not marvelous). Unless there is a flaw I haven’t spotted, I could live with it.

(4) ‘‘Ultra light’’

Here is an absolutely minimal (and non-standard) allocator that Hans Bohm has been playing with:

```
        // basic allocator, all allocators must provide these functions:

        class alloc {
        public:
                static void * allocate(size_t positive_number_of_bytes);
                static void deallocate(void *non_null_pointer,
                                       size_t requested_size_of_first_arg);
                static void * reallocate(void *original_object,
                                       size_t old_sz, size_t new_sz);
        };

        // example of a user-provided allocator for type T:

        template<class T, class alloc> class simple_alloc {
        public:
                static T *allocate(size_t n = 1)
                        { return (T*) alloc::allocate(n * sizeof (T)); }
                static void deallocate(T *p, size_t n = 1)
                        { alloc::deallocate(p, n * sizeof (T)); }
                // ...
```

```
        };

Pro:
        Alternative allocators.
Con:
        No alternative pointers.
        No alternative accessors.
Observation:
        Our designs haven't had a realloc(). I cannot rate its utility
        in the context of allocators for containers. One reason, I use
        containers is exactly to avoid writing calls to realloc().
        All members are static for maximal efficiency. Does an allocator
        object really need to carry information?
My rating:
        Too radical departure this late in the game. Good basic idea.
        Good efficiency. Can be integrated into something like the current
        allocator scheme. I'd miss alternative accessors and I suspect
        the lack of alternative accessors would bit us/someone.

(5) ''Light''

What would it take to support alternative allocators, alternative pointers,
and alternative accessors without imposing overheads compared to the ''ultra
light'' solution and without making it hard to use like the ''status quo?''
In addition, a solution should be close to ''status quo'' notationally
wherever possible.

In more detail, what do we (in my opinion) want to be able to do with
an allocator:

        the usual typedefs

        allocate N (uninitialized) objects of type T (an arbitrary type T)
        delete them

        allocate N (initialized) objects of type T (an arbitrary type T)
        delete them

        construct an allocated object
        destroy an object without deallocating it

The current WP allocator does that, but somehow gets tangled in void*s
and somewhat obscure "rebinding." The pre-Stockholm also seems to do that.

        allow alternative pointer types/"models"

        allow container operations to take and return "pointer" and
        "reference" types for a type X rather that X* and X&.

        hide the decision to use "pointer" and "reference" type from
        the container operations through an allocator.

The pre-Stockholm seems to do that also.

What would the minimal class that did that look like?

First I take status quo and throw away rebind. The typedefs seem
minimal and appropriate. So does the functions - provided we don't
constrain the typedefs or get tied up in knots when allocating objects
of "other" types. Then I take the 'static' from the "ultra-light"
approach (to force someone to explain why an allocator object needs
to carry data). Finally, I use yet another technique for providing
an allocator for arbitrary types as part of an allocator (it is
a remote cousin of the rebind struct).

template<class T> class allocator {
```

```
public:
        typedef T value_type;
        typedef size_t size_type;
        typedef ptrdiff_t difference_type;

        typedef T* pointer;
        typedef const T* const_pointer;

        typedef T& reference;
        typedef const T& const_reference;

        allocator();
        ~allocator();

        static pointer address(reference) const;
        static const_pointer address(const_reference) const;

        static pointer allocate(size_type, const_pointer hint = 0);
        static void deallocate(pointer);

        static size_type max_size() const;

        static void construct(pointer p, const_reference val);
        static void destroy(pointer p);

        template<class T2> class other : public allocator<T2> { }
};
```

Now I can write the canonical difficult example with what looks like minimal fuss:

```
template<class T, class A = allocator<T> >
class list {
        // ...
        iterator insert(iterator position, const T& x = T())
        {
                typename A::other<link>::pointer p = A::other<link>::allocate(1);
                A::pointer q = A::allocate(1);
                // ...
        }
}
```

If - as I think we all assume - allocation of objects of "other" types is rare the notational overhead is acceptable, and as usual a typedef could be used:

```
        typedef typename A::other<link> link_alloc;
        typedef typename A::other<link>::pointer link_ptr;

        link_ptr p = link_alloc::allocate(1);
```

Alternative allocators can be written as specializations od allocator or separately in the style of allocator.

Pro:

- Alternative allocators.
- Alternative pointers.
- Alternative accessors.
- Simple user interface.
- Efficient.
- Close to status quo.

Con:

- You didn’t see it before Stockholm.

My rating:

- Best, but it needs more thought than status quo got before Stockholm.

- Bjarne