STL Allocator using Partial Specialization

by Nathan Myers <myersn@roguewave.com>
Rogue Wave Software

26-Jan-95
X3J16/95-0019
WG21/N0619

1. Will Member Class Templates Survive?

At Valley Forge we accepted a form of STL allocator which was based on a language feature, member class templates, accepted and added to the WP in November 1993. Some implementors mentioned at Valley Forge that they had not noticed that member class templates had been accepted, and, further, that they consider it impractical to implement. They have promised to come to Austin with a proposal to remove the feature, and others have promised to demonstrate an implementation of the feature.

It is necessary for the Library not to depend on a feature removed from the language. Fortunately, the run-time variable allocator facility can be recast in terms of partial specialization, which all implementors seem to agree can be implemented (more or less) easily, and which users find more generally useful in any case. A proposal to add partial specialization to the language is expected at Austin.

Part 1 of this proposal is to either cancel the previous proposal (94-161R2/N0548R2), if member class templates are removed and partial specialization is not accepted, or to change the form of the allocator interface if member class templates are removed and partial specialization is accepted. The following text details the latter case.

The following definitions are provided in a standard header:

```
  namespace std {

  template <class T, class Allocator>
  class pointers {
    typedef T* ptr;
    typedef T const* cptr;
    typedef T& ref;
    typedef T const& cref;
    typedef T value_type;
  };

  template <class Allocator>
  class pointers<void, Allocator> {
   public:
    typedef void* ptr;
    typedef void value_type;
  };
```

...along with the default allocator, which may simply delegate to
the global operator new (but may perform optimization of its own):

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

    template <class T>
      pointers<T,allocator>::ptr
      address(pointers<T,allocator>::ref x) const
        { return &x; }
    template <class T>
      pointers<T,allocator>::cptr
      address(pointers<T,allocator>::cref x) const;
        { return &x; }

    allocator() {}
    ~allocator() {}

    template <class T>  // for use by garbage collectors...
      void destroy(pointers<T>::ptr p) { p->~T(); }

    template <class T, class U>
      pointers<T,allocator>::ptr
      allocate(size_type howmany, pointers<U,allocator>::cptr hint)
        { return operator new(howmany * sizeof(T)); }

    template <class T>
      void
      deallocate(pointers<T>::ptr p)
        { operator delete(p); }

    size_type max_size() const;
  };

  inline void* operator new(size_t N, allocator& a)
    { return a.allocate(N, 0); }

  } // namespace std
```

The members allocate() and deallocate() are parameterized to allow
them to be specialized for particular types in user allocators. Users
(e.g. object database vendors) may also define allocators of their
own with compatible interfaces.

It is assumed that any pointer types have a (possibly lossy)
conversion to void*, yielding a pointer sufficient for use as
the "this" value in a constructor or destructor, and conversions
to (and from?) pointers<void,A>::ptr (for appropriate A) as well,
for use by A::deallocate().

Collections are parameterized exactly as in the previous proposal.

2. Member allocator::init_page_size()

As a separate issue, (regardless of whether partial specialization

is used) I propose to remove the member

```
  class allocator {
    ...
    template <class T> size_type init_page_size() const;
  };
```

as obsolete. I have ascertained that it was intended for use by collections to manage blocks of storage elements; this is now the responsibility of allocator itself, or of specializations of the member allocator::allocate<T,U>(...).

3. Name Changes in Allocator

Regardless of whether member class templates are removed from the language, some name changes seem in order. In place of the name, e.g. allocator::types<T>::pointer, I recommend allocator::pointers<T>::ptr, and so on:

```
  class allocator {
    ...
    template <class T>
    class pointers {
      typedef T* ptr;
      typedef T const* cptr;
      typedef T& ref;
      typedef T const& cref;
      typedef T value_type;
    };
    ...
  };
```

The name "types" is perhaps too generic, and its few member names need not be so long to be distinguished from one another.