Doc. No.:   WG21/N1000=X3J16/96-0182
Date: 23 Sep 1996
Project: C++ Standard Library
Reply to: Nathan Myers
            <ncm@cantrip.org>

Clause 20 (Utilities Library) Issues (Revision 6)

** Revision History:

```
   Revision 0 - 22 May 1995 [was Version 1]
   Revision 1 - 09 Jul 1995 [was Version 2] (edits before Monterey)
   Revision 2 - 26 Sep 1995 (pre-Tokyo)
   Revision 3 - 30 Jan 1996 (pre-Santa Cruz)
   Revision 4 - 28 May 1996 (pre-Stockholm)
   Revision 5 - 09 Jul 1996 (Stockholm)
   Revision 6 - 23 Sep 1996 (pre-Kona)
```

** Introduction

This document is a summary of issues identified for the Clause 20, identifying resolutions as they are voted on, and offering recommendations for unsolved problems in the Draft where possible.

```
** Work Group:     Library: Utilities Clause 20
** Issue Number:   20-032
** Title:          Allocator pointer and reference required conversions
                   need clarification
** Sections:       20.1.4 [lib.allocator.requirements]
** Status:         active
```

** Description:

(This is Box 20-1 in the pre-Stockholm draft.)
The table of Allocator requirements specifies conversions:

```
  X::pointer  -->  T*, void*, X::const_pointer, XT<void>::const_pointer
  X::const_pointer  -->  T const*, void const*, XT<void>::const_pointer
  X::reference -->  T&
  X::const_pointer  -->  T const&
```

and describes the conversions to built-in pointers and references as yielding a value suitable to use as "this" in a member function. The conversion to XT<void>::const_pointer (which is shorthand for X::rebind<void>::other::const_pointer) is for use as the "hint" argument to allocate.

The question is, is this a complete set of necessary conversions, or does the list require refinement? In particular, should some reference conversions (e.g. X::reference --> X::const_reference) be required as well?

** Proposed Resolution:

(none yet)

```
** Requestor:      Nathan Myers <ncm@cantrip.org>
** Owner:
```

```
** Work Group:     Library: Utilities Clause 20
** Issue Number:   20-039
** Title:          definitions incomplete
```

** Sections: 20.1.1, 20.1.2, 20.1.3
** Status: active

** Description:

The definition of EqualityComparable in 20.1.1 uses the term
"equivalence relationship." This term is not defined within the
draft.

The definition of LessThanComparable in 20.1.2 uses the term
"total ordering relationship." This term is not defined within the
draft.

Does "equivalence" include constness? The second line of the
CopyConstructible requirements in 20.1.3, table 40 requires u
(where u is on object of const T) to be equivalent to T(u). T(u)
will not return a const object.

** Discussion:
** Proposed Resolution:
(none yet)

** Requestor: John Benito
** Owner:

** Work Group: Library: Utilities Clause 20
** Issue Number: 20-040
** Title: 20.1.4 paragraph 2 unclear
** Sections: 20.1.4
** Status: active

** Description:

20.1.4 p2 states that requirements of types manipulated through
allocators are specified in table (32). If this table does somehow
contain this information, it is in no way clear.

** Discussion:

I’m not sure what John is getting at here.

** Proposed Resolution:
(none yet)

** Requestor: John Benito
** Owner:

** Work Group: Library: Utilities Clause 20
** Issue Number: 20-041
** Title: allocator operators new[], delete[] need count argument
** Sections: 20.4.1.2 [lib.allocator.globals]
** Status: active

** Description:

The global

```
  template <class T>
    void operator delete[](void*, allocator<T>& a);
```

is not implementable, because it needs an (unavailable) element
count argument to pass to a.deallocate().   This can only be provided

```
by an argument, which implies that a matching argument is required for

  template <class T>
    void* operator new[](size_t, allocator<T>&);

because normally operator delete[] is called only by the exception
handling runtime system, which gets the arguments for operator delete[]
from the arguments passed to operator new[].

Also, in the description, the value passed in the size_t argument was
wrong, and the number of arguments to deallocate was off.

** Discussion:
** Proposed Resolution:

Replace the declarations and descriptions of the above operators,
in the synopsis 20.4 [lib.memory], and in the description 20.4.1.2
[lib.allocator.globals] as follows:

  template <class T>
    void* operator new[](size_t bytes, allocator<T>& a, size_t count);

  Requires: bytes == count*sizeof(T)
  Returns:  a.allocate(count);
  Notes:    invocation as "new(a,N) T[N]" results in correct arguments.


  template <class T>
    void operator delete(void* p, allocator<T>& a);

  Requires: p obtained by calling a.allocate, not yet deallocated.
  Effect: a.deallocate(p,1)


  template <class T>
    void operator delete[](void* p, allocator<T>& a, size_t count);

  Requires: p obtained by calling a.allocate, not yet deallocated.
  Effect: a.deallocate(p,count)

Furthermore, editorially:

Prototypes for operators == and != appear incorrectly in 20.4 [lib.memory]
but correctly in 20.4.1 [lib.default.allocator].  The correct declarations
should be moved to replace the incorrect declarations.

Finally, operator new is declared in both 20.4 and 20.4.1.  It should be
removed from 20.4.1.

** Requestor:      Nathan Myers <ncm@cantrip.org>
** Owner:

---------------

** Work Group:     Library: Utilities Clause 20
** Issue Number:   20-042
** Title:          auto_ptr<> assignment semantics disastrous
** Sections:
** Status:         active

** Description:

My comments below will refer to the following extract of the working
paper of the draft (in [lib.auto.ptr], p 20-26 of the "DRAFT: 20
September 1996"):
```

```
--------
1:  template<class Y> auto_ptr& operator=(const auto_ptr<Y>& a) throw();
2:
3:  Requires:  Y is type  X or a class derived from  X for which    delete
4:    (Y*) is defined and accessible.
5:  Effects:  If  *this owns  *get() and  *this != &a then  delete  get().
6:    Calls  a.release().
7:  Returns:   *this.
8:  Postconditions:   *this holds the pointer returned from   a.release().
9:     *this owns  *get() if and only if a owns  *a.
```

Problem 1: an auto_ptr target of a self-assignment dangles. This is because of the unconditional call to a.release() on line 6.

Problem 2: there is a leak in some situations such as:

```
        p = q;
        p = q;
```

After the first assignment, both p and q point to the same resource, but p owns it. This ownership -and thus any ownership of the resource- is lost during the second assignment.

Similar cases are met every time when, in an assignment "p = q;", p and q both point to the same resource, but p owns it.

** Discussion:

The ambiguity about the behaviour shows in the following slip:

```
5:   Effects:  If  *this owns  *get() and  *this != &a then  delete  get().
                                           ^
```

Should one read: "this != &a" or "this->get() != a.get()"?

I argue that this behaviour is not acceptable, for several reasons:

- This behavior loses compared to plain pointers (loss of safety!).
- It results -in my example above- from the fact that the assignment modifies its const argument through the use of mutable, in a way that is not anymore semantically free for the user.
  I claim that this is an abuse of "mutable".
- The current behavior is surprising for users, even if it results from a simple specification.
- The main area of use for auto_ptrs is likely to be the writing of simple exception safe code. What is expected of "exception safe" code is mostly code that doesn’t leak in presence of exceptions. It is therefore especially critical to prevent this kind of problems.

This is an argument to change the assignment semantics as described below.

Example of implementation:

```
   template<class Y> auto_ptr& operator=(const auto_ptr<Y>& r) throw() {
      if ((void*)&r != (void*)this) { // I didn’t understand Greg’s test
         if (px != r.px)
            if (owner)
               delete px;
            owner = r.owner;
            px = r.px;

         } else
            owner |= r.owner; // don’t transfer non-ownership to owner
         r.owner = 0;
      }
      return *this;
   }
```

** Proposed Resolution:

Change the description of auto_ptr<> assignment semantics as follows:

```
Effects: If this == &a, skip.
  Otherwise, consider two cases: *this and a hold pointers to
  different resources, or to the same resource.
  In the first case, if *this owns *get, delete get(); then transfer
  the ownership and the pointer.
  In the second case, take care that the ownership is not lost.
  In both cases, insure that a doesn't own the pointer.

Postconditions: *this holds the pointer returned from a.get(). *this
  owns *get() if and only if either of *this or a owned it previously.
```

```
** Requestor:      Marc Girod <marc.girod@ntc.nokia.com>  (FINLAND)
** Owner:
```

```
** Work Group:     Library: Utilities Clause 20
** Issue Number:   20-043
** Title:          Discussion of const conversions omitted from auto_ptr<>
** Sections:
** Status:         active
```

** Description:

On line 3 (in issue 22-042), shouldn't the draft mention the case of
const conversion?:

```
3:  Requires:  Y is type  X or a class derived from  X for which    delete
                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

In this example, Y is neither X, nor a class derived from X:

```
  auto_ptr<const X> p;
  auto_ptr<X> q;
  p = q;
```

** Discussion:

The same would apply to the copy constructor.

** Proposed Resolution:
(no WP text offered)

```
** Requestor:      Marc Girod <marc.girod@ntc.nokia.com>
** Owner:
```

```
** Work Group:     Library: Utilities Clause 20
** Issue Number:   20-044
** Title:          Template copy constructors not used implicitly
** Sections:       20.4.1 [lib.default.allocator], 20.4.5 [lib.auto.ptr]
** Status:         active
```

** Description:

Now that the definition of template constructors has been clarified,
we now recognize that the definitions of many of the library components
is incomplete and incorrect.

In particular, those components which are defined with a template
constructor in a form similar to a copy constructor, e.g.:

```
  template <class T>
    class X {
      ...
      template <class U>
        X(const X<U>&);  // a copy constructor?
      ...
  };
```

do not actually have a copy constructor defined, and the language
description is such as to require that a copy constructor be generated,
not as an implicit specialization of the declared template constructor,
but rather according to the rules for automatic generation of ordinary
undeclared copy constructors.

** Discussion:

In the Draft, we have generally not mentioned copy constructors for
classes for which the semantics or declaration would be identical
to that generated automatically. Exceptions to this are those cases
where copy semantics is abnormal (e.g. in auto_ptr<>) or where more
stringent requirements are imposed (e.g. a nil throw() specification,
as in allocator<>).

We should add explicit declarations of these constructors to the
descriptions of those class templates to which the above remarks
apply.

** Proposed Resolution:
(none yet)

```
** Requestor:      Steve Rumsby <steve@maths.warwick.ac.uk>
** Owner:
```

```
** Work Group:     Library: Utilities Clause 20
** Issue Number:   20-xxx
** Title:
** Sections:
** Status:         active

** Description:
** Discussion:
** Proposed Resolution:

** Requestor:
** Owner:
```

Closed issues:

```
** Issue Number:   20-001
** Title:          Allocator needs operator ==
** Resolution:     passed

** Issue Number:   20-002
** Title:          allocator::types<> has no public members
** Resolution:     passed
** Issue Number:   20-003
** Title:          Allocator requirements incomplete
** Resolution:     passed

** Issue Number:   20-004
** Title:          allocator parameter "hint" needs hints on usage
** Resolution:     passed

** Issue Number:   20-005
** Title:          Default allocator member allocate<T>() doesn’t "new T".
** Resolution:     passed

** Issue Number:   20-006
** Title:          allocator::max_size() not documented
** Resolution:     passed

** Issue Number:   20-007
** Title:          C functions asctime() and strftime() use global locale
** Status:         closed

** Issue Number:   20-008
** Title:          construct() and destroy() functions should be members
** Resolution:     passed

** Issue Number:   20-009
** Title:          Allocator member init_page_size() no longer appropriate.
** Resolution:     closed

** Issue Number:   20-010
** Title:          auto_ptr specification wrong.
** Status:         passed

** Issue Number:   20-011
** Title:          specialization of allocator::types<void> incomplete
** Resolution:     passed

** Issue Number:   20-012
** Title:          get_temporary_buffer has extra argument declared
** Resolution:     passed

** Issue Number:   20-013
** Title:          get_temporary_buffer semantics incomplete
** Resolution:     passed

** Issue Number:   20-014
** Title:          allocator could be a template again
** Status:         passed

** Issue Number:   20-015
** Title:          class unary_negate ill-specified.
** Resolution:     passed

** Issue Number:   20-016
** Title:          binder{1st|2nd}::value types wrong.
** Resolution:     passed

** Issue Number:   20-017
** Title:          implicit_cast template wanted
** Status:         closed, no action (Tokyo)

** Issue Number:   20-018
** Title:          auto_ptr::reset to self
** Status:         closed, implemented choice 2 (Tokyo)

** Issue Number:   20-019
** Title:          no default ctors on many lib classes
** Status:         closed, no action (Tokyo)

** Issue Number:   20-020
** Title:          Template constructor for pair<>
** Status:         passed

** Issue Number:   20-021
** Title:          should pair<> have a default constructor?
** Status:         closed, implemented (Tokyo)

** Issue Number:   20-022
** Title:          unary_compose and binary_compose missing.
** Status:         closed, no action (Tokyo)

** Issue Number:   20-023
** Title:          pair<> should have typedefs
** Status:         closed, implemented

** Issue Number:   20-024
** Title:          pointer_to_unary/binary_function pass-by-value
** Status:         passed

** Issue Number:   20-025
** Title:          Stack, queue, and priority_queue adaptor templates should
                   not have allocator parameter.
** Status:         passed

** Issue Number:   20-026
** Title:          raw storage iterators and others described in
                   terms of nonexistent components.
** Status:         passed

** Issue Number:   20-027
** Title:          allocator new and delete incomplete
** Status:         passed

** Issue Number:   20-028
** Title:          auto_ptr<> need throw() specifications
** Status:         passed

** Issue Number:   20-029
** Title:          General pointer comparisons needed for use in set<>, map<>.
** Status:         passed

** Issue Number:   20-030
** Title:          auto_ptr<> descriptions improperly imply undefined behavior
** Status:         closed

** Issue Number:   20-031
** Title:          Function object "times" collides with common C function name
** Status:         passed


** Issue Number:   20-033
** Title:          allocator::address members need clarification
** Status:         closed

** Issue Number:   20-034
** Title:          Use of "hint" argument to allocate need clarification
** Status:         passed

** Issue Number:   20-035
** Title:          Allocator requirements table typo cleanup
** Status:         passed
** Issue Number:   20-036
** Title:          Complexity specifications meaningless?
** Status:         passed

** Issue Number:   20-037
** Title:          Allocator::deallocate needs count argument
** Status:         passed

** Issue Number:   20-038
** Title:          class allocator specialization for void has extra members
** Status:         passed
```