

Doc. No.: WG21/N0937=X3J16/96-0119  
Date: 28 May 1996  
Project: C++ Standard Library  
Reply to: Nathan Myers  
<ncm@cantrip.org>

Clause 20 (Utilities Library) Issues (Revision 4)

\*\* 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)

\*\* 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.

-----

\*\* Issue Number: 20-007  
\*\* Title: C functions asctime() and strftime() use global locale  
\*\* Sections: 20.5 [lib.date.time]  
\*\* Status: closed by default (Tokyo)

\*\* Description:

From Box 8 in the pre-Stockholm draft:

Note: in Monterey we accepted the resolution for issue 20-007 in 95-0099R1, the body of which was "to be specified"! So this sub-clause still needs work :-)

\*\* Discussion

20-007 concerned the relationship between C functions asctime() etc. and the C++ locale. Now that we know there is none, I am leaving this closed, and recommending that the Editor remove the box.

\*\* Requestor: Steve Rumsby

-----

\*\* Work Group: Library: Utilities Clause 20  
\*\* Issue Number: 20-024  
\*\* Title: pointer\_to\_unary/binary\_function pass-by-value  
\*\* Sections: 20.3.7 Adapters for pointers to functions  
          [lib.function.pointer.adaptors]  
\*\* Status: active

\*\* Description:

operator() of pointer\_to\_unary\_function and pointer\_to\_binary\_function currently pass their arguments by const reference. This prevents ptr\_fun() from working with functions which pass by const reference because the pointer\_to.. operator() arguments end up being references to references. For example:

```
int my_fun(const Foo& bar) { ... }
```

```
for_each( ..., ..., ptr_fun( my_fun ) ); // oops! error!
```

This problem has been fixed in the HP STL distribution by changing the pointer\_to.. operator() arguments to pass by value, so this represents status quo in the outside world.

**\*\* Proposed Resolution:**

In 20.3.7 [lib.function.pointer.adaptors] class pointer\_to\_unary\_function, change:

```
Result operator()(const Arg& x) const;
to:
Result operator()(Arg x) const;
```

In 20.3.7 [lib.function.pointer.adaptors] class pointer\_to\_binary\_function, change:

```
Result operator()(const Arg1& x, const Arg2& y) const;
to:
Result operator()(Arg1 x, Arg2 y) const;
```

**\*\* Requestor:** Beman Dawes  
**\*\* Owner:**

-----  
**\*\* Work Group:** Library: Utilities Clause 20  
**\*\* Issue Number:** 20-025  
**\*\* Title:** Stack, queue, and priority\_queue adaptor templates should not have allocator parameter.

**\*\* Sections:**  
**\*\* Status:** active

**\*\* Description:**

(This really should be a Clause 23 issue, but it also concerns allocator)

The stack<>, queue<>, and priority\_queue<> adaptor templates as currently defined take a template parameter Allocator, which is not used by the adaptor except as an argument to the constructor. This allocator can and should be obtained from the Container argument class.

**\*\* Proposed Resolution:**

Eliminate the Allocator template parameter in each of stack<>, queue<>, and priority\_queue<>. Change constructor arguments of these templates to declare an argument using the member typedef from the Container argument, like:

```
stack(typename Container::allocator_type = Container::allocator_type());
```

**\*\* Requestor:** Bjarne Stroustrup  
**\*\* Owner:**

-----  
**\*\* Work Group:** Library: Utilities Clause 20  
**\*\* Issue Number:** 20-026  
**\*\* Title:** raw storage iterators and others described in terms of nonexistent components.

**\*\* Sections:**  
**\*\* Status:** active

**\*\* Description:**

Michael writes:

Was the exclusion of an Allocator template parameter in the raw\_storage\_iterators, destroy() and the uninitialized\_\*() algorithms, and possibly the temporary\_buffer algos simply an oversight? The draft defines their behavior based on functions that have since been removed.

\*\* Proposed Resolution:  
(none yet)

\*\* Requestor: mklobe@objectspace.com  
\*\* Owner:

-----  
\*\* Work Group: Library: Utilities Clause 20  
\*\* Issue Number: 20-027  
\*\* Title: allocator new and delete incomplete  
\*\* Sections: 20.1.4 [lib.allocator.requirements],  
20.4 [lib.memory],  
20.4.1.2 [lib.allocator.globals]  
\*\* Status: active

\*\* Description:

Clause 20 defines operator new() for allocators, but does not define operator new[](), nor operators delete() and delete[](). These should be added to the table of allocator requirements and to the class interface for the default allocator, and incorporated into the examples.

These are needed so that the memory will be deallocated if an exception is thrown from a constructor.

\*\* Proposed Resolution:

Add to the Allocator requirements table in [lib.allocator.requirements]:

```
operator delete(void* p, x)      (none)      x.deallocate(p)
operator delete[](void* p, x)   (none)      x.deallocate(p)
```

Add to the <memory> synopsis in [lib.memory] and the default allocator globals in [lib.allocator.globals]:

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

Requires: p obtained by a call to allocator<T>::allocate, not yet deallocated.  
Effect: x.deallocate(p).

\*\* Requestor: mklobe@objectspace.com  
\*\* Owner:

-----  
\*\* Work Group: Library: Utilities Clause 20  
\*\* Issue Number: 20-028  
\*\* Title: auto\_ptr<> need throw() specifications  
\*\* Sections:  
\*\* Status: active

\*\* Description:

In lib-4686, Greg Colvin:

I was recently reminded that since I first proposed `auto_ptr` the restrictions on exception handling (`lib.res.on.exception.handling`) in the WP have changed from:

- 1 Any of the functions defined in the C++ Standard library can report a failure to allocate storage by throwing an exception of type `bad_alloc`, or a class derived from `bad_alloc`.
- 2 Otherwise, none of the functions defined in the C++ Standard library throw an exception that must be caught outside the function, unless explicitly stated otherwise.

to:

- 1 Any of the functions defined in the C++ Standard library can report a failure by throwing an exception of the type(s) described in their Throws: paragraph and/or their exception-specification (`_except.spec_`). An implementation may strengthen the exception-specification for a function by removing listed exceptions.
- 2 None of the functions from the Standard C library shall report an error by throwing an exception, unless it calls a program-supplied function that throws an exception.
- 3 Any of the functions defined in the C++ Standard library that do not have an exception-specification may throw any exception. An implementation may strengthen this implicit exception-specification by adding an explicit one.

Therefore `auto_ptr` now needs exception specifications.

An `auto_ptr` requires no free store, and requires nothing of its type argument but an accessible delete operation. The delete operation is used in `~auto_ptr()`, so `~auto_ptr()` can throw anything thrown by the delete. No other `auto_ptr` function need throw anything.

\*\* Proposed Resolution:

All operations on `auto_ptr` but its destructor should have a `throw()` specifications:

```
namespace std {
  template<class X> class auto_ptr {
  public:
    // 20.4.5.1 construct/copy/destroy:
    explicit auto_ptr(X* p=0) throw();
    template<class Y> auto_ptr(const auto_ptr<Y>&) throw();
    template<class Y> auto_ptr& operator=(const auto_ptr<Y>&) throw();
    ~auto_ptr();
    // 20.4.5.2 members:
    X& operator*() const throw();
    X* operator->() const throw();
    X* get() const throw();
    X* release() const throw();
  };
}
```

\*\* Requestor: Greg Colvin <greg@imrgold.com>  
\*\* Owner:

-----

\*\* Work Group: Library: Utilities Clause 20  
\*\* Issue Number: 20-029  
\*\* Title: General pointer comparisons needed for use in `set<>`, `map<>`.  
\*\* Sections:  
\*\* Status: active

\*\* Description:

Standard containers `set<>` and `multiset<>` depend on a total ordering among elements. If pointers are to be stored in these structures, a means is needed to provide a total ordering on pointers. The language does not provide such an operation.

Lengthy reflector mail advanced three alternatives:

1. Extend, in the core language, `operator<` applied to pointers to yield a total order;
2. Provide a library function object `less_pointer(void*, void*)` defined to yield a total order for pointers;
3. Require that the standard function objects `less<>` etc., specialized on pointers, yield a total order. They would be explicitly specialized only on architectures for which `operator<` does not already provide a total order.

**\*\* Discussion:**

For (1), The LWG has no say in language core extensions. The Core group consensus seems to be against such an extension, on the grounds that the performance impact on some architectures could be prohibitive.

Alternative (2) adds yet another component to the library, or more if `greater_pointer<>`, `less_equal_pointer<>`, etc are also added.

Alternative (3) reduces flexibility for users who might be storing pointers within a single array, and who do not want to incur the expense of general pointer comparisons when built-in `operator<()` suffices.

(2) is less convenient to use than (3); if (3) is implemented, users can avoid the total ordering expense by defining their own comparison object.

**\*\* Proposed Resolution:**  
Add to `[lib.comparisons]`

For templates `greater`, `less`, `greater_equal`, and `less_equal`, the specializations for any pointer type yield a total order, even if built-in operators `<`, `>`, `<=`, `>=` do not.

**\*\* Requestor:** Beman Dawes  
**\*\* Messages:** Core-6691, 6649, 6650, 6651.  
**\*\* Owner:**

-----  
**\*\* Work Group:** Library: Utilities Clause 20  
**\*\* Issue Number:** 20-030  
**\*\* Title:** `auto_ptr<>` descriptions improperly imply undefined behavior  
**\*\* Sections:**  
**\*\* Status:** active

**\*\* Description:**

Our WP specifies that violations of Requires clauses give undefined behavior. However, the Requires clauses in 20.4.5.1 (`auto_ptr` ctors) can all be diagnosed at compile time. I can see no benefit to not requiring these diagnostics, so some small changes are in order.

**\*\* Proposed Resolution:**

Change:

20.4.5.1 auto\_ptr constructors  
[lib.auto\_ptr.cons]

```
explicit auto_ptr(X* p = 0);
```

Requires:

p points to an object of type X or a class derived from X for which  
^^^^^^

| p shall point

delete p is defined and accessible, or else p is a null pointer.

...

```
template<class Y> auto_ptr(const auto_ptr<Y>& a);
```

Requires:

Y is type X or a class derived from X for which delete(Y\*) is defined  
^^^^

| Y shall be

and accessible.

...

```
template<class Y> auto_ptr<X>& operator=(const auto_ptr<Y>& a);
```

Requires:

Y is type X or a class derived from X for which delete(Y\*) is defined  
^^^^

| Y shall be

and accessible.

\*\* Requestor: Greg Colvin

\*\* Owner:

-----

\*\* Work Group: Library: Utilities Clause 20

\*\* Issue Number: 20-031

\*\* Title: Function object "times" collides with common C function name

\*\* Sections: 20.3 [lib.function.objects] and  
20.3.2 [lib.arithmetic.operations]

\*\* Status: active

\*\* Description:

I would like to see the name of the function object "times" which performs multiplication i.e. template <class T> struct times changed to "multiplies".

The reason for this is that the name "times" conflicts with the XPG4 "times" function declared in <sys/times.h>. Eventually this conflict should be resolved by putting STL times in the std namespace. However renaming "times" to "multiplies" will prevent confusion for people who are familiar with the XPG4 times routine and it clearly identifies the function of the STL routine.

\*\* Proposed Resolution:

In [lib.function.objects] and in [lib.arithmetic.operations],

Change the template type name "times" to "multiplies".

\*\* Requestor: Sandra Whitman <whitman@tle.enet.dec.com>

\*\* Owner:

-----

\*\* 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: 20-033  
\*\* Title: allocator::address members need clarification  
\*\* Sections: 20.4.1, 20.4.1.1 [lib.default.allocator],  
[lib allocator.members]  
\*\* Status: active

\*\* Description:

Members address() are defined to apply operator& to the reference argument. This leaves unclear whether built-in or member operator& is used. We should make it clear that the built-in operator is used, and that no exception is thrown.

\*\* Proposed Resolution:

For the default allocator members allocator::address():

```
pointer address(reference x) const;  
const_pointer address(const_reference x) const;
```

add "throw()" to each, and document that they return "::operator&(x)", not "&x".

\*\* Requestor: Nathan Myers  
\*\* Owner:

-----

\*\* Work Group: Library: Utilities Clause 20  
\*\* Issue Number: 20-034  
\*\* Title: Use of "hint" argument to allocate need clarification  
\*\* Sections: 20.4.1.1 [lib.allocator.members]  
\*\* Status: active

\*\* Description:

Box 6 in the pre-Stockholm draft:

TBS: using "hint" should be documented as unspecified, but intended as an aid to locality if an implementation can use it so.

\*\* Discussion:

In the Allocator Requirements ([lib.allocator.requirements], 20.1.4):

6 The second parameter to the call a.allocate in the table above is an implementation-defined hint from the container implementor to the allocator, typically as an aid for locality of reference.

with the footnote:

In a container member function, the address of an adjacent element is often a good choice to pass for this argument.

This describes Allocator semantics well enough for implementors of containers, but may not say enough about the default allocator.

I don't know what "implementation-defined", in paragraph 6, means in that context.

\*\* Proposed Resolution:

In [lib.allocator.requirements], paragraph 6 quoted above, strike "implementation-defined".

In [lib.allocator.members], add to allocator::allocate():

Requires: \*hint\* either 0 or previously obtained from member allocate and not yet passed to member deallocate. The value \*hint\* may be used by an implementation to help improve performance.

\*\* Requestor:

\*\* Owner:

-----

\*\* Work Group: Library: Utilities Clause 20  
\*\* Issue Number: 20-035  
\*\* Title: Allocator requirements table typo cleanup  
\*\* Sections: 20.1.4 [lib.allocator.requirements]  
\*\* Status: active

\*\* Description:

The description of required Allocator member x.construct is:

x.construct(p,u) (not used) Effect: new((void\*)p) T(u)

but u is not a value of type T, but a pointer to a value of type T. This is a typo. Also, the line



```
typename X::
rebind<U>::other      for an instantiation
                      of XT<T>, the type XT<U>
```

uses an undefined name, XT. This should be in the previous table.

\*\* Proposed Resolution:

Change the Table 6 "construct" entry to:

```
x.construct(p,t)  (not used)      Effect: new((void*)p) T(t)
```

Add to Table 5:

```
XT<T>           same as X
```

and change the definition of rebind to:

```
typename X::
rebind<U>::other      the type XT<U>
```

```
** Requestor:      Nathan Myers
** Owner:
```

-----

```
** Work Group:      Library: Utilities Clause 20
** Issue Number:    20-0xx
** 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: 20-008  
\*\* 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