Doc. No.: X3J16 / 95-0099
WG21 / N0699
Date: 22 May 1995
Project: C++ Standard Library
Reply to: Nathan Myers
myersn@roguewave.com

Clause 20 (Utilities Library) Issues (V.1)

Revision History:

Version 1 - 22 May 1995

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:	20-001
Title:	Allocator needs `operator==()`
Sections:	20.1, 20.4.1
Status:	active

Description:

Allocator is a value class, passed to container and string constructors and copied into them for use in managing any secondary storage they use. When assigning one object to another (particularly e.g. string) that use reference-counting to share storage, it is necessary to be able to determine that the same space is used; e.g. the strings are in the same database.

The Allocator requirements as specified to not allow this.

Discussion:

Now that Allocator is a full object, it needs the usual "nice" qualities: copy operators and comparison for equality.

Proposed Resolution:

Add to Table 18-2:

Expression	Return Type	assert/Pre/Post
a1 == a2	bool	Returns true iff the two allocators are interchangeable, such that storage allocated from each can be deallocated via the other
a1 != a2	bool	same as !(a1 == a2)
a1 = a2	X&	post: a1 == a2
X a1(a2);		post: a1 == a2

Define a global operator (in std::):

```
inline bool operator==(const allocator&, const allocator&)
{ return true; }
```

for the default allocator in 22.4.1.

Requestor:
Owner:

Work Group: Library: Utilities Clause 20
Issue Number: 20-002
Title: `allocator::types<>` has no public members
Sections: 20.4.1
Status: active
Description:

The member class template `allocator::types<>` has no public members. I think this is editorial, because the enabling proposal specified all public members.

Discussion:

I believe this was just a typo.

Proposed Resolution:

Either add "`public:`" to the definition, or change it to a struct, at the editor's discretion. (I prefer a struct.)

Requestor:
Owner:

Work Group: Library: Utilities Clause 20
Issue Number: 20-003
Title: Allocator requirements incomplete
Sections: 20.1 and 20.4.1
Status: active

Description:

Many of the requirements on Allocator, in general, are mentioned only in the section describing the default allocator. In particular, every Allocator type needs a member class template types<> which provides public type members, and member functions appropriate to them.

Discussion:

The proposed resolution below merely completes the incorporation of Allocator requirements already accepted.

Proposed Resolution:

1. Replace the first three rows in Table 18-2 as follows:

Expression	Return Type
typename X::types<T>::pointer	convertible to T* and void*
typename X::types<T>::const_pointer	convertible to const T* and to const void*
typename X::types<T>::reference	convertible to T&
typename X::types<T>::const_reference	convertible to const T&
typename X::types<T>::value_type	Identical to T

> .and the "allocate" and "deallocate" rows, where X is an Allocator type, x and y have type X&, y has type const X&, p, q, r, and s are values of type X::types<T>::pointer, const_pointer, reference, and const_reference, respectively, for any type T, and u has type X::types<U>::pointer for any type U.

Expression:	Return Type:
x.template address<T>(r)	X::types<T>::pointer
x.template address<T>(s)	X::types<T>::const_pointer
x.template allocate<T,U>(n,u)	X::types<T>::pointer
x.template deallocate<T>(p)	(not used)

Expression:	Return Type:
new(x) T	new((void*)x.template allocate<T,void>(1, 0)) T
new(x) T[n]	new((void*)x.template allocate<T,void>(n, 0)) T[n] (n>0)

(Adopt the same "assertions / conditions" text from the existing table.)
Also, add:

> a. a precondition on the parameter to deallocate<>: its argument must have been obtained by calling some y.allocate where (x == y). b. a note that the result of allocate() is a Random Access Iterator.

2. Move paragraph 2 from 20.4.1 up to section 20.1, after table 20-2.

Requestor:
Owner:

Work Group: Library: Utilities Clause 20
Issue Number: 20-004
Title: allocator parameter "hint" needs hints on usage
Sections: 20.1
Status: active
Description:

The Draft contains no text to explain how the "hint" parameter to the Allocator member template `allocate<>()` is used.

Discussion:

This parameter was added at the request of OODB vendors who have found that the availability of such a hint can lead to orders of magnitude better performance.

Proposed Resolution:

Add a paragraph in 20.1 after Table 20-2:

> The second parameter to the call x.template allocate<> in the table is a hint. For best performance, it should be a pointer to an object typically used about the same time as the object being allocated, but it could be a null pointer if necessary. In a member function, "this" is usually a good choice to use.

Requestor:
Owner:

Work Group: Library: Utilities Clause 20
Issue Number: 20-005
Title: Default allocator member `allocate<T>()` doesn't "new T".
Sections: 20.4.1.1
Status: active

Description:

From the Draft:

```
      template<class T, class U>
            typename types<T>::pointer
            allocate(size_type n, typename
types<U>::const_pointer hint);
```

Notes:
Uses ::operator new(size_t) (_lib.new.delete_).
Returns:
new T, if n == 1. Returns new T[n], if n > 1.
+------- BEGIN BOX 2 -------+
ISSUE: Is this right? How does deallocate() know which form of delete to use?

The member allocate doesn't call constructors, so it can't use "new T" or "new T[n]". It has to call "operator new(...)" directly.

Discussion:

I worry that we are missing an opportunity to permit substantial optimizations by specifying that `allocate()` and `deallocate()` explicitly call operators `new()` and `delete()`. The sizes and types of the objects involved are lost in the translation.

Proposed Resolution:

I see two choices:

1. Replace the description above with:

Returns:
operator new(n * sizeof(T))
Throws:
bad_alloc if the amount of memory requested is not available.

This is simplest, but leaves no opportunity for optimization.

2. Replace the descriptions of both default allocator members allocate and deallocate to indicate that they manage memory obtained in an unspecified manner, in the same sense as global operators `new()` and `delete()`. Furthermore, identify these function templates as "replaceable" in the same sense as are global operators `new()` and `delete()`.

Requestor:
Owner:

Work Group: Library: Utilities Clause 20
Issue Number: 20-006
Title: `allocator::max_size()` not documented
Sections: 20.4.1.1
Status: active
Description:

In the Allocator requirements, `max_size()` is specified to return the largest positive value of `difference_type`. For the default allocator, this is `ptrdiff_t`.

Proposed Resolution:

Document `allocator::max_size()` as:

Returns: numeric_limits<ptrdiff_t>::max(), where ptrdiff_t is as found in the header <cstddef>.

Requestor:
Owner:

Work Group: Library: Utilities Clause 20
Issue Number: 20-007
Title: C functions `asctime()` and `strftime()` use global locale
Sections: 20.5
Status: active
Description:

The Draft describes the functions `asctime()` and `strftime()` as identical to the C Library functions of the same name. However, they depend on the global locale, which is not the same in the C++ Library. We need text here to describe how they use the global locale.

Discussion:

The mapping is quite straightforward: they use

```
use_facet< time_put<char> >(locale()).put(...)
```

as many times as necessary to format their results.

Proposed Resolution:
[TBS]
Requestor:
Owner:

Work Group: Library: Utilities Clause 20
Issue Number: 20-008
Title: `construct()` and `destroy()` functions should be members
Sections: 20.4, 20.4.3, 20.1, 20.4.1
Status: active
Description:

The Draft provides several functions of dubious value:

```
template <class T> T*   allocate(ptrdiff_t n, T*);
template <class T> void deallocate(T* buffer);
template <class T1, class T2> void construct(T1* p, const T2&
value);
template <class T>            void destroy(T* pointer);
```

These were useful in building the HP STL library, which did not use standard Allocators; however, anyone building a standard Container object could not use them, but would instead use the Allocator interface.

Discussion:

The functions above do not aid communication between modules or provide substantial functionality, or even serve users as a good example; hence, they do not meet the normal criteria for inclusion in the Library.

However, functions like them proved convenient in building the STL. Rather than remove them entirely, I proposed last time adding them to the standard Allocator interface, and was asked to return with a full proposal.

Proposed Resolution:

1. Eliminate the functions mentioned above from 20.4 and 20.4.3.

2. Add to Table 20-2:

and t has type const T& ...

Expression:	Result Type:	Assert/Pre/Post:
x.template construct<T,U>(p,u)	(not used)	Effect: new((void*)p) T(u)
x.template destroy<T>(p)	(not used)	Effect: ((T*)p)->~T()

3. Add to the default allocator, in section 20.4.1, member function templates:

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

both defined as in the table.

4. Add notes to front matter in Clauses 21 (Strings) and 23 (Containers) that specify that all storage components described retain between calls to their member functions comes from a copy of the allocator passed to their respective constructors.

Requestor:
Owner:

Work Group:	Library: Utilities Clause 20
Issue Number:	20-00
Title:	Allocator member `init_page_size()` no longer appropriate.
Sections:	20
Status:	active

Description:

In the HP STL implementation, before allocator, collections did their own bulk storage management. Now that we have allocator objects that can be tuned (and, if necessary, specialized) for the purpose, this function is not so useful, and indeed encourages a harmful practice.

This member was omitted from the latest Draft, but not (to my knowledge) by any enabling motion. We should make it official.

Discussion:

Encouraging collections to perform bulk storage management is incompatible with best performance in object databases. Furthermore, such code obscures the logic of containers, and tends to be identical or nearly so in each container, resulting in wasteful duplication. Such code was intended to reduce the number of calls to the (expensive) global operator new(), but such optimization would better be performed in the allocator.

Proposed Resolution:
A choice:

1. No changes; simply ratify the omission in the existing draft.

2. Reinstate `init_page_size()`.

Requestor:
Owner:
