Doc No: WG21 N4002
Date: 2014-05-21
Reply to: Nicolai Josuttis (nico@josuttis.de)
Subgroup: LWG
Prev. Version: none

# Cleaning-up noexcept in the Library

With N3279 we introduced some guidelines for how to use noexcept in the C++ Standard Library. However, after some years of experience, we learned that we have to

- Improve these guidelines
- Fix places where we agree that according to the old and new guidelines things are, or may be a problem
  Discuss and decide whether (or not) to fix issues upon which we have no clear opinion

# Motivation for this Paper

The noexcept guidelines used for C++11 are essentially as follows:

- Each library function, having a **wide** contract (i.e. does not specify undefined behavior due to a precondition), that the LWG agree **cannot throw**, should be marked as **unconditionally noexcept**.
- If a library **swap** function, **move** constructor, or **move** assignment operator ... can be proven not to throw by applying the noexcept operator then it should be marked as **conditionally noexcept**. No other function should use a conditional **noexcept** specification.
- No library destructor should throw. It shall use the implicitly supplied (non-throwing) exception specification.
- Library functions designed for compatibility with C code ... may be marked as unconditionally noexcept.

However, it turned out that we have two issues with this topic:

a) We have to fix the noexcept handling for containers and strings
b) We might change these guidelines

Let me discuss these two topics separately.

## Necessary noexcept fixes for noexcept in Containers and Strings

One question that came up with issue 2319 was how to deal with exceptions that might be thrown in move constructors in debug mode. In Issaquah we decided therefore to remove noexcept for the move constructor of std::string with C++17.

Note that it is *only* the **move constructor** that is problematic here; **move assignment** can (and probably always will) degenerate to a copy if almost any stateful allocator is used, which leads to a conditional noexcept as discussed later.

The goal was not to remove noexcept entirely. So, one option raised was to mark these functions (and others) as “highly recommended to be noexcept” without requiring it. But then, we need a way to signal this in the Standard.

In a discussion on the library reflector about this (“introducing "normative encouragement to not throw exceptions"), there was a change in opinions, so that we now

- agree to have `noexcept` declarations for string and vector move constructors
- and have the need to discuss, whether to declare move constructors of other containers as `noexcept`

One reason was that using `noexcept` can affect performance by a factor of 10 in some example programs, such as the following example by Howard Hinnant (with some modifications):

```
#include <vector>
#include <string>
#include <chrono>
#include <iostream>
using namespace std;
using namespace std::chrono;

class X
{
  private:
    string s;
  public:
    X()
     : s(100, 'a') {
    }

    X(const X& x) = default;

    X (X&& x) NOEXCEPT
     : s(move(x.s))
    {
    }
};

int main()
{
  vector<X> v(1000000);
  cout << "cap.: " << v.capacity() << endl;

  auto t0 = high_resolution_clock::now();
  v.emplace_back();
  auto t1 = high_resolution_clock::now();

  auto d = duration_cast<milliseconds>(t1-t0);
  cout << d.count() << " ms\n";
}
```

Another observation was that when defining the move constructor as noexcept, then usually also the default constructor can be defined as `noexcept` because (as STL stated):

> Note that default ctors and move ctors are twins when it comes to noexcept - either both should be marked, or neither. This is nearly a fundamental law - if an object always needs to acquire a resource even in its default-constructed state, then the move ctor also needs to acquire such a resource (because you start with one object and end with two), in order to avoid emptier-than-empty. But if an object can be default constructed noexceptly, then move construction can be implemented with default construction and nofail swap.

However, as Howard Hinnant pointed out:

> I agree there is a close relationship here as Stephan describes. There is a caveat here though. I can not find anywhere in the allocator requirements that if the allocator is default_constructible, that it is nothrow_default_constructible. We have two choices:
> 1. Require that allocators be either !is_default_constructible<A>{} || is_nothrow_default_constructible<A>{}. or:
> 2. vector{} is noexcept only if Allocator{} is noexcept. [Note: std::allocator{} is already noexcept].
>
> I prefer 2. It gives allocator authors more latitude for negligible cost.
> Also we currently specify vector{} like so:
> vector() : vector(Allocator()) { }
> It would be so much better to specify it with:
> vector() noexcept(is_nothrow_default_constructible<allocator_type>{});
> I.e. Not require (nor even encourage) an allocator copy construction.

Comment on that by STL:

> As allocator copies and moves are forbidden from throwing (17.6.3.5 [allocator.requirements]), I dislike the approach here. I would like to see allocator default construction, if present, to be forbidden from throwing. (Whether copies, moves, and default ctors should be detected as noexcept by the type traits is a separate question.) Then basic_string and vector's default ctors can be unconditionally noexcept.

Note, however, that we already require in
**17.6.3.5 Allocator requirements [allocator.requirements]:**

> No constructor, comparison operator, copy operation, move operation, or swap operation on these types shall exit via an exception.

The default constructor is a constructor. Thus, **we already require that the default constructor, move constructor, and move assignment operator never throw exceptions**.
John Lakos comments on this as follows:

> I would suggest that we (at least) consider relaxing this wording to allow for Howard's suggestion about having default construction of allocators NOT to necessarily be treated as **noexcept**, and making container constructors be conditionally **noexcept,** based on that compile-time property. (Note that, for the kind of allocators we routinely deal with in practice, just like our own vectors and strings, it isn't a practical issue the way it might be for node-based containers).

However, this is a different issue, which I don’t propose with this paper.

## Handling Different Allocators

One question that came up while we discussed the whole problem is what to do if we have move **assignments** where the objects use different allocators:

- If the allocator type is different, the string/container type is different, so there is no problem.
- However, with scoped or other stateful allocators the type might be the same while the instance of the allocator is different. In this case:
  - Allocators of the same type may have different states,
  - which means that the move assignment sometimes has to copy elements,
  - which means that the move assignment might throw.

So, for move assignment (that is where two different allocators might appear), we need a conditional noexcept, resulting to false, if the allocator instances might have different states. For that case we need to know whether the allocator is interchangeable. Thus we need something yielding

- true for the default allocator,

- but returning false for stateful allocators (such as polymorphic allocators)

Note that this issues is proposed and discussed already with:

http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html#2108

We discussed again different alternatives:

a) Directly checking whether the allocator class is empty, which would signal that it has no state. But the state might be a table outside the class.
b) Adding a trait signaling whether allocator instances are interchangeable (always return true for operator ==).
c) Another trick, suggested in issue 2108, is to let operator == for allocators return `true_type` when it is always true and then use traits to check whether allocators operator == return type is such a type or just a bool.

In this paper I prefer option b). That is, I don't propose the trick proposed in issue 2108 because IMO the trick with true_type is not easy to understand, which might lead to more errors than it solves. I prefer to provide a more intuitive and self-explaining approach (we have several other places where we require operations not to contradict each other).

I also suggest to use `is_always_interchangeable` instead of `always_compares_equal`.

So, we propose might a new allocator trait is_always_interchangeable, which returns true_type if the allocator is always interchangeable (i.e. operator == for this allocator type always will return true).

Roughly, the default would be:

typedef is_empty<allocator> is_always_interchangeable;

which is fine for all allocators in the Standard. You would (and should) only have to overwrite this value for an allocator if you have state members but are still interchangeable or have no state members but a state. Thus, you can overwrite this in either direction.

Note, however, that also POCMA (propagate on container move assignment) is involved here:

- If POCMA is true, we do not need to detect mismatched allocators. Then we can simply adjust pointers, without any potential for throwing.
- If POCMA is false, we need to compare allocators for equality.  If equal, adjust pointers (can't throw).  If non-equal, we have to allocate a memory chunk and move elements into it, and behave as if their move ctors might throw (for vector; string elements are POD).

Thus, for move assignments we propose the following noexcept condition:

```
allocator_traits<Allocator>::propagate_on_container_move_assignment::value
|| allocator_traits<Allocator>::is_always_interchangeable::value
```

This is roughly what Howard Hinnant proposes in http://stackoverflow.com/questions/12332772/why-arent-container-move-assignment-operators-noexcept with the different to use is_always_interchangeable and || instead of &&.

Note that Pablo Halpern wrote:

> I wonder if we need this trait at all, or if we can just change the default definition of POCMA to:
>
> `is_empty<X>`
>
> The noexcept clause for vector and string would then simply be:
>
> ```
> noexcept(allocator_traits<Allocator>::propagate_on_container
> _move::value)
> ```

## Summary of Proposed Changes

- For allocators_traits:
  - Introduce a new allocator trait is_always_interchangeable
    with corresponding entry in allocator requirements
- For existing allocators:
  - Add a specific definition for is_always_interchangeable
- For string:
  - Cancel the proposed resolution of issue 2319 to remove noexcept for the move constructor of std::basic_string.
  - For move assignment, add a compile-time check to determine whether the allocators are either interchangeable or POCMA is true..
- For vector:
  - Add unconditionally noexcept to the default constructor
  - Add unconditionally noexcept to the move constructor
  - Add conditionally noexcept to the move assignment operator as for strings.

**Open**:

Relation to the following other library issues:

http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html#2016

covers: Allocators must be no-throw swappable

http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html#2063

covers: string move assignment fixes

http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html#2152

covers: swap for containers

http://www.open-std.org/jtc1/sc22/wg21/docs/lwg-active.html#2321

Moving containers should (usually) be required to preserve iterators

## Wording of Proposed Changes

(all against N3937)

### In **17.6.3.5 Allocator requirements [allocator.requirements]**

in Table 28 after propagate_on_container_swap (at the end) add table entry:

Expression:
`X::is_always_interchangeable`

Return type:
Identical to or derived from `true_type` or `false_type`

Assertion/note Default pre-/post-condition:
`true_type` if the expression `x1 == x2` is guaranteed to be true for any two (possibly `const`) values `x1`, `x2` of type `X`, when implicitly converted to `bool`. See Note B, below.

Default:
`is_empty<X>`

And after Note A, add:

Note B: If `X::is_always_interchangeable::value` or `XX::is_always_interchangeable::value` evaluate to `true` and an expression equivalent to `x1 == x2` or `x1 != x2` for any two values `x1`, `x2` of type `X` evaluates `to false` or `true`, respectively, the behavior is undefined.

### In **20.7.8 Allocator traits [allocator.traits]**

in struct allocator_traits:

after:

`typedef` *see below* `propagate_on_container_swap;`

add:

`typedef` *see below* `is_always_interchangeable;`

### In **20.7.8.1 Allocator traits member types [allocator.traits.types]**

after §9 (before rebind_alloc) add:

`typedef` *see below* `is_always_interchangeable;`

*Type:* `Alloc::is_always_interchangeable` if the qualified-id `Alloc::is_always_interchangeable` is valid and denotes a type (14.8.2 [temp.deduct]); otherwise `is_empty<Alloc>`.

### In **20.7.9 The default allocator [default.allocator]**

in class allocator

after:

```
typedef true_type propagate_on_container_move_assignment;
```

add:

```
typedef true_type is_always_interchangeable;
```

In **20.13.1 Header <scoped_allocator> synopsis [allocator.adaptor.syn]**

in class scoped_allocator_adaptor:

After:

```
typedef see below propagate_on_container_swap;
```

add:

```
typedef see below is_always_interchangeable;
```

In **20.13.2 Scoped allocator adaptor member types [allocator.adaptor.types]**

After §4 (propagate_on_container_swap)

add:

```
typedef see below is_always_interchangeable;
```

> *Type*: `true_type` if `allocator_traits<A>::is_always_interchangeable::value` is `true` for **every** `A` in the set of `OuterAlloc` and `InnerAllocs`...; otherwise, `false_type`.

In **21.4 Class template basic_string [basic.string]**

in `class std::basic_string`

Replace:

```
basic_string() : basic_string(Allocator()) { }
```

by

```
basic_string() noexcept : basic_string(Allocator())  { }
```

Unlike issue 2319, keep

```
basic_string(basic_string&& str) noexcept;
```

Replace

```
basic_string& operator=(basic_string&& str) noexcept;
```

by

```
basic_string& operator=(basic_string&& str)
noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value
           || allocator_traits<Allocator>::is_always_interchangeable::value);
```

In **23.3.6.1 Class template vector overview [vector.overview]**

in `class std::vector`

Replace

```
vector() : vector(Allocator()) { }
```

by

```
vector() noexcept : vector(Allocator())  { }
```

Replace

```
vector(vector&&);
```

by

```
vector(vector&&) noexcept;
```

Replace

```
vector& operator=(vector&& x);
```

by

```
vector& operator=(vector&& x)
noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value
           || allocator_traits<Allocator>::is_always_interchangeable::value);
```

## Things to Discuss

a) Should other containers have the same noexcept specifications as vector now?

b) Should other containers get a “strong recommendation to be noexcept”

c) If b), what should the recommendation look like:
Proposal: `[[noexcept]]`

Note: Conforming implementations may add noexcept, but not remove noexcept (according to [res.on.exception.handling]/1):

> "An implementation may strengthen the exception-specification for a non-virtual function by adding a non-throwing noexcept-specification."

## Possible Fixes to the noexcept guidelines

From the fixes above the following changes are probably useful:

Possible additional guideline:

- If a move constructor has a (conditional) noexcept specification, the default constructor should have the same (conditional) noexcept specification.
- If for objects allocators are involved and the move assignment operator does not throw with the default allocator, the corresponding move assignment operator should have a conditional noexcept specification as follows:

```
allocator_traits<Allocator>::propagate_on_container_move_assignment::value
           || allocator_traits<Allocator>::is_always_interchangeable::value);
```

As Pablo Halpern pointed out:

> Important to distinguish between move construction and move assignment.

Also, Peter Dimov pointed out:

> In my opinion, the current wide/narrow practice is wrong.
>
> It's wrong on a conceptual level, because (almost) no function is actually wide. All functions have implicit requirements that their arguments, *this, and everything else reachable from them be a valid object. (Or, in the case
> of a constructor, that 'this' points to storage suitable to hold an object.)
>
> It's also wrong because it sets up a conflict. When specifying, say, operator*, we now need to make a choice between adding a Requires clause and a noexcept, the two being mutually exclusive under the wide/narrow theory. This does not improve the quality of the specification.
>
> I understand the motivation for all that. The idea is that the requirements are asserted, and the fact that the requirements are asserted is tested by making the assertions throw. But somehow I don't find this sufficient cause to degrade the specification of the standard library for everyone.

If I understand Peter Dimov’s comment correctly, saying a wide contract is defined if no precondition fails leads to a situation where we never have a wide contract because there are several preconditions that are implicitly and not explicitly (by Requires paragraphs) defined.
One solution might be, that we define that we have a wide contract if there is no requires paragraph (see also [res.on.required]). Something along the following lines:

- Each library function having a wide contract that does not have a requires paragraph [res.on.required] and where, that the LWG agrees that it cannot throw should be marked as unconditionally noexcept.
- If a library swap function, move constructor, or move-assignment operator is conditionally wide (i.e. can be proven to not throw by applying the noexcept operator specification/condition) then it should be marked as conditionally noexcept. No other function should use a conditional noexcept specification.

Other comments:

Note that we are disobeying the guideline “No other function should use a conditional **noexcept** specification” in non-member cbegin()/cend(), for "convenience" (this handles arrays).

Otherwise we do appear to be strictly following this convention, at least in N3936 (didn't check the TS).

So we might strike the sentence “No other function should use a conditional noexcept specification.”

## Acknowledgments

Thanks to all committee members discussion this issue and Pablo Halpern, Howard Hinnant, Daniel Krügler, John Lakos, Stephan T. Lavavej, Jonathan Wakely, and some guys at C++Now for the work on the proposed paper and wording.