Doc. No: X3J16/97-0039
WG21/N1077
Date: 30 May 1997
Project: Programming
Language C++

Reply to: Matt Austern
austern@sgi.com

STANDARD LIBRARY EXCEPTION POLICY

SUMMARY

Clause 17 currently says that the library is not exception-safe: if the library calls user code, and the user code throws an exception, then you get undefined behavior.

This is a proposal to change that. The core idea is that a library class or function is required to be exception-safe if the user-defined
classes and functions that it invokes are themselves exception-safe.

DEFINITIONS

A function f is exception safe if
(1) If an exception is thrown from within the function, it will not cause a resource leak.
(2) Any object that f modifies will remain in a stable state, even if an exception is thrown from within f.

A class X is exception safe if
(1) X::~X() doesn't ever throw an exception.
(2) If an exception is thrown from within one of X's constructors,
it won't cause a resource leak. Any resources that were acquired prior to the exception being thrown will be released.
(3) All of X's member functions and friend functions are exception safe.

A function is commit-or-rollback if it is exception safe and, additionally, it makes the following guarantee:
If an exception is thrown from within f, then the state of all objects that f modifies will be restored to the state they had
before f was called.

[NOTE: A consequence of these definitions is that if a non-member function f doesn't allocate any resources, and if the types that it operates on are all exception safe, then f is automatically exception safe as well.]

DISCUSSION

I have now implemented a version of the STL in which all classes and functions are exception safe, *provided* that the template arguments provided by the user are exception safe. (And also provided that the template parameters obey a few other constraints.) Some of the classes and functions, but by no means all, have commit-or-rollback

semantics. There are some member functions where commit-or-rollback would be either impossible or prohibitively expensive.

Here's what we need to do with the standard.

(1) Change the semantics of some of the low-level library components in Clause 20. (It's effectively impossible to write exception-safe code using those components as they are currently defined.

(2) Specify the exact requirements on user classes and functions. This should go in Clause 17, since it's a blanket requirement for the entire library.

(3) Get rid of the sentence in Clause 17 saying that throwing an exception from within a library component results in undefined behavior.

In my opinion, there is only one serious issue that's open for discussion: should we say that the library is exception safe, and be done with it, or should we attempt to explicitly list every member function that is not only exception-safe but also commit-or-rollback?

I favor the former. Here are my reasons.

(1) Fewer changes to the WP. A blanket statement in one place is much better than statements scattered throughout the library clauses.

(2) Insufficient time. At this point, I believe that if we tried to generate a list like that we would probably not get it right.

(3) Insufficient experience. I've looked at, and implemented, exception safety in the STL portion of the standard library. I haven't touched some of the other parts of the library, though. To be blunt, we are talking about standardization in advance of implementation experience; it pays to be very conservative.

(4) Avoidance of overconstraint. I could come up with a list of functions that I have implemented to be commit-or-rollback, but it's not at all obvious that that list would be suitable as requirements for every conforming implementation.

[Note that there's nothing stopping vendors from documenting which of the functions and classes in their implementations are commit-or-rollback. None of these four objections applies to vendor-specific documentation.]

RESTRICTIONS ON USER CODE

These restrictions are essentially those of "Vectors and Exceptions" X3J16/97-0019 = WG21/N0157. At the time I wrote it, I had only studied vectors in detail; I have now implemented exception safety in the entire STL, though, so we can be reasonably confident that these restrictions are necessary and sufficient for the entire library.

[Note that violating these restrictions does not necessarily result in undefined behavior; it results in undefined behavior only if an exception is thrown from within a standard library component. And, again, note that "undefined behavior" gives vendors license to implement and document library components that are robust even when the user-provided code violates these restrictions.]

A. General restrictions.

All functions invoked by the library, and all types provided as template parameters for library templates, must be exception-safe.

B. Restrictions on allocators.

If A is an allocator type, then neither A::deallocate() nor A::destroy()
may ever throw an exception.

If Ptr is a pointer type (A::pointer or A::const_pointer), then valid pointer operations on objects of type Ptr (that is, operations where the preconditions specified in the Random Access Iterator requirements
table are satisfied) may never throw exceptions.

[Note: this wording is awkward because it is intended to permit objects of type Ptr to throw exceptions in two circumstances. (1) When preconditions are not satisfied---an attempt to dereference a null pointer, for example. (2) If Ptr has member functions that are not found in the Random Access Iterator requirements, then those operations may throw exceptions.]

In the version of this proposal that I posted to the reflector, I also
included the requirement that operations on A::size_type and A::difference_type may not throw exceptions. I've removed that requirement, because Sean pointed out that it's redundant: built-in types, like int and long, don't throw exceptions. (size_type and difference_type are required to be "integral types", and the phrase "integral type" has a specific definition in clause 3.)

CHANGES TO LOW-LEVEL LIBRARY COMPONENTS

A. Specialized algorithms.

As it stands, the three specialized algorithms in 20.4.4 are useless for implementing exception-safe containers. If an exception is thrown
from somewhere within one of those algorithms, then exception safety, even in a very weak sense, demands that the recovery code determine how many constructor calls succeeded before the exception was thrown. With the interface currently specified, this is impossible.

The simplest change is to require that all three of these functions be
commit-or-rollback. That is: if there is an exception, then all objects constructed before the exception was thrown will be destroyed.
This turns out to be very easy to implement.

This implies one more restriction: iterator operations on the arguments of these three functions may not throw exceptions.

B. Temporary buffers.

Get_temporary_buffer and return_temporary_buffer are not suitable for implementing exception-safe algorithms; a far better solution is a temporary_buffer class.

Since every library implementor will need some class of this sort, and
since users who want to implement exception-safe adaptive algorithms will also need it, the most sensible choice is to add it to the standard. I do not propose removing get_temporary_buffer and

return_temporary_buffer, however; they're redundant, but there's no reason to gratuitously break programs that use them.

```
       template <class ForwardIterator,
                 class T =
iterator_traits<ForwardIterator>::value_type>
       class temporary_buffer {
       public:
         temporary_buffer(ForwardIterator first, ForwardIterator
last);
         ~temporary_buffer();

         T* begin();
         T* end();
         ptrdiff_t size() const;
         ptrdiff_t requested_size() const;
       private:
         temporary_buffer(const temporary_buffer&) {}
         void operator=(const temporary_buffer&) {}
       };
```

[Note: This is a very minimal interface, but it suffices for exception-safe adaptive generic algorithsm. Alex Stepanov and I, and others, have written many adaptive algorithms using this interface.]

SPECIFIC WP CHANGES

All WP changes are in clause 17 and clause 20. I am leaving out the changes to the allocator requirements: I'm writing a separate proposal
to deal with allocators, and it makes much more sense to list all allocator-related WP changes together.

CHANGES IN CLAUSE 17

Add the following text at the end of section 17.1 [definitions]

-- stable state. An object's state is stable if none of its member or friend functions, when called with arguments that satisfy their preconditions, result in undefined behavior. [Note: in particular, this applies to the object's destructor.] An object's state is unstable if it is not stable.

-- exception safe function. A function f is exception safe if an exception thrown from within f does not cause a resource leak
(objects that are constructed and never destroyed, or memory that is allocated and never deallocated) and does not cause the state of any object that f modifies to become unstable.

-- exception safe class. A class X is exception safe if all of its member and friend functions are exception safe, and, additionally, its destructor is guaranteed never to throw an
exception.

-- commit-or-rollback. A function f is commit-or-rollback if it is exception safe and, additionally, it guarantees that, if an exception is thrown from within f, all objects that f modifies will be restored to the same states as before f was called.

Remove the last sentence from paragraph 2 of section 17.3.3.6 [lib.res.on.functions]. That is, remove the text

> **-- if any of these functions or operations throws an exception, unless specifically allowed in the applicable Required Behavior paragraph.**

Add the following text after paragraph 1 of section 17.3.4.8 [lib.res.on.exception.handling]

> **In certain cases (replacement functions, handler functions, operations on types used to instantiate standard library template components), the C++ Standard Library depends on components supplied by a C++ program. Unless explicitly stated in the applicable Required behavior paragraph, these components may report failure by throwing exceptions.**
>
> **All standard library components are exception safe. If the standard library depends on a component supplied by a C++ program, that component is required to be exception safe. If the library depends on a component that is not exception safe, and an exception is thrown by either the library or the supplied component, the resulting behavior is undefined.**

CHANGES IN CLAUSE 20

Add class temporary_buffer to the header <memory> synopsis in section 20.4 [lib.memory].

Add the following text after paragraph 1 of section 20.4.4 [lib.specialized.algorithms].

> **All of the following algorithms are commit-or-rollback. In all of the algorithms, the formal template parameter ForwardIterator is required to have the property that no exceptions are thrown from increment, assignment, comparison, or dereference of valid iterators.**

Add the following text, as a new section, after section 20.4.3 [lib.temporary.buffer]. Number the new section 20.4.4, and renumber sections 20.4.4 through 20.4.6 accordingly.

> **20.4.4 Temporary buffer class**
>
> **Template class temporary_buffer allocates temporary storage, and deallocates the storage when it itself is destroyed. It does not provide copy or assignment semantics.**

```
namespace std {
   template <class ForwardIterator,
             class T =
iterator_traits<ForwardIterator>::value_type>
   class temporary_buffer {
   public:
     temporary_buffer(ForwardIterator first, ForwardIterator
last);
     ~temporary_buffer();

     T* begin();
     T* end();
     ptrdiff_t size() const;
```

```
      ptrdiff_t requested_size() const;

   private:
     temporary_buffer(const temporary_buffer&) {}
     void operator=(const temporary_buffer&) {}
   };
}
```

20.4.4.1 temporary_buffer requirements

The formal template parameters ForwardIterator and T are required to satisfy the following properties.

- ForwardIterator conforms to the requirements of a mutable forward iterator.
- T has a constructor that can take a single argument of type iterator_traits<ForwardIterator>::value_type.
- T has an accessible destructor.

20.4.4.2 temporary_buffer constructor and destructor

temporary_buffer(ForwardIterator first, ForwardIterator last)

Requires: [first, last) is a valid range.
Effects: Creates a temporary buffer that contains at most distance(first, last) objects of type T. The initial values of the buffer's elements are arbitrary. [Note: this gives the implementation freedom to use whichever of T's constructors is most convenient or efficient.]

~temporary_buffer()

Effects: Destroys all of the objects in the buffer, and deallocates the buffer's storage.

20.4.4.3 temporary_buffer members

T* temporary_buffer::begin()
Effects: Returns a pointer to the first element in the buffer.

T* temporary_buffer::end()
Effects: Returns a pointer one past the the last element in the buffer.

ptrdiff_t requested_size() const
Effects: Returns the buffer's requested size. That is, it returns distance(first, last), where first and last are the arguments that were used to construct the buffer.

ptrdiff_t temporary_buffer::size() const
Effects: Returns the number of elements in the buffer. [Note: size() == end() - begin().] The returned value satisfies the constraint 0 <= size() <= requested_size().