Doc. No: X3J16/97-0019
WG21/N0157
Date: 11 March 1997
Project: Programming Language C++
Reply to: Matt Austern
austern@sgi.com

VECTORS AND EXCEPTIONS

The exception policy for the standard library is still an open issue. What guarantees can we make on how library components behave when user code throws exceptions, and what restrictions on user code do we need to impose in order to provide those guarantees?

At this point, the only standard container that I have examined in detail is vector<>. Based on a cursory examination, though, my guess is that the situation with the other containers is similar.

Summary: we can achieve a fairly high degree of exception safety for vector<T, Allocator>. Specifically, we can provide these guarantees.

(1) Even if user code throws an exception, the vector will still be in a state where it can be destroyed safely.
(2) Exceptions won't result in resource leaks. All of the vector's elements that have been properly constructed will eventually be destroyed.
(3) The vector will be in some sane state after an exception--- not necesarily a predictable or useful state, but it will still be a vector. The program won't crash if you call any of its member functions.
(4) vector<>::~vector() doesn't throw an exception.

These guarantees hold if we make the following restrictions on vector<T, Allocator>'s template parameters.

(1) T's destructor doesn't throw an exception under any circumstances.
(2) No operations on T, including assignment, can ever put an object of class T into a non-destructable state.
(3) Allocator::deallocate may not throw an exception.
(4) Objects of type Allocator::pointer, Allocator::const_pointer, Allocator::size_type, and Allocator::difference_type may not throw exceptions.

Additionally, we need a guarantee from uninitialized_copy, uninitialize_fill, and uninitialized_fill_n. The guarantee we need is in fact full rollback capability: if anything in one of those algorithms throws an exception, then any objects that have been constructed will be destroyed. These three algorithms can provide that guarantee, provided that we also impose some restrictions on their arguments.

(1) The template arguments that are iterators or integral types can't throw exceptions upon valid iterator arithmetic or dereference of a valid iterator.
(2) The value type's destructor can't throw an exception.

Here the detailed analysis follows. I'm using the SGI implementation (of course), but I don't think that the details would be remarkably different if I were looking at a different implementation.

Just as a guide, vector<> has three member variables: start, finish, and end_of_storage. It has two important invariants: (1) Either all three members are null, or else they are all valid

pointers. In the latter case, start <= finish <= end_of_storage.
(2) Every element in [start, finish) is a valid T, and the range
[finish, end_of_storage) consists entirely of uninitialized storage.

I'm only looking at non-const member functions.

CONSTRUCTORS

```
1: vector() : start(0), finish(0), end_of_storage(0) {}
```

If pointers are of some user-defined type, then any of these three
null pointer constructors might throw an exception. This should
be harmless, however.

```
1: vector(size_type n, const T& value) {
2:   start = data_allocator::allocate(n);
3:   uninitialized_fill_n(start, n, value);
4:   finish = start + n;
5:   end_of_storage = finish;
6: }
```

The allocation in line 2 might fail. This is harmless, since
there isn't yet anything else to be destroyed.

The uninitialized_fill_n in line 3 is more interesting: it might fail
after some of the vector's elements have been constructed. If we
don't want resource leaks, then we need to destroy the
already-constructed elements. That means we have to keep track of
which elements have been constructed and which have not been. The
most reasonable place to do that is within uninitialized_fill_n: it
must guarantee that, if an exception is thrown anywhere within it, all
constructed objects must be destroyed.

The pointer assignments in lines 4 and 5, and the pointer arithmetic
in line 4, might fail if pointer is a user-defined type. If we
don't want resource leaks, we'll have to catch the exception thrown
by the pointer and destroy all of the already-constructed elements.
We might use code that goes something like this.

```
1: try {
2:   finish = start + n;
3:   end_of_storage = finish;
4: }
5: catch (...)
6:   destroy(start, start + n);
7:   data_allocator::deallocate(start, start + n);
8:   throw;
9: }.
```

There's a problem with that, though. The call to destroy in line 6
uses just the same operations that we are worried about in lines 2 and
3. (And it actually uses one more operation than that: it has to
dereference the pointers that it is passed as arguments.) Conclusion:
we can't recover from an exception thrown in the course of pointer
arithmetic, assignment, and dereference. User-defined pointer types
should not be permitted to throw exceptions.

One other interesting point about user-defined pointer types.
If there is an exception, we want to deallocate vector's storage if
and only if the allocation succeeded. There are two ways to
do this. First, have a separate try block around the call
data_allocator::allocate(n). Second, initialize start to be a null
pointer. In that case, we will know in the catch block that
the allocation succeeded if and only if start is non-null. The latter

scheme is simpler, and it also fits better with handling zero-sized vectors. This implies that user-defined pointer types must support null pointers.

```
1: vector(size_type n) {
2:   start = data_allocator::allocate(n);
3:   uninitialized_fill_n(start, n, T());
4:   finish = start + n;
5:   end_of_storage = finish;
6: }
```

This constructor raises no new issues.

```
1: vector(const vector<T, Alloc>& x) {
2:   start = data_allocator::allocate(x.end() - x.begin());
3:   finish = uninitialized_copy(x.begin(), x.end(), start);
4:   end_of_storage = finish;
5: }
```

The same analysis as above applies: uninitialized_copy must have the same properties as uninitialized_fill_n.

```
1: vector(const_iterator first, const_iterator last) {
2:   size_type n = 0;
3:   distance(first, last, n);
4:   start = data_allocator::allocate(n);
5:   finish = uninitialized_copy(first, last, start);
6:   end_of_storage = finish;
7: }
```

DESTRUCTOR

```
1: ~vector() {
2:   destroy(start, finish);
3:   deallocate();
4: }
5:
6: void deallocate() {
7:   if (start) data_allocator::deallocate(start, end_of_storage - start);
8: }
```

The only way to be sure that there are no resource leaks, and that vector<>::~vector() doesn't throw any exceptions, is if every operation in the destructor is guaranteed to succeed. This means that the call to destroy() and the call to deallocate() must succeed. The destroy() algorithm iterates through a range (in this case, a range of pointers), calling the destructor for each element *i. Deallocate() uses the vector's allocator to deallocate storage. This implies the following restrictions.

(1) User-defined pointer types may not throw exceptions when a pointer in a valid range is incremented.
(2) A valid pointer may not throw an exception when it is dereferenced.
(3) An allocator's deallocation function may not throw an exception.

BASIC ACCESSOR FUNCTIONS

```
1: iterator begin() { return start; }
2: iterator end() { return finish; }
3: reference operator[](size_type n) { return *(begin() + n); }
4: reference front() { return *begin(); }
5: reference back() { return *(end() - 1); }
```

At first sight, these member functions raise no new issues. In fact, however, they do. These functions give the user access to the vector's elements; if the user performs an operation on a vector element that puts that element in an inconsistent (non-destructable) state, then the vector itself will also be in an inconsistent state.

Consider, for example, *(V.begin()) = x. If the assignment fails halfway through, and the first element in V is in an inconsistent state, then attempting to destroy V could cause a crash.

Note that the only vector<> member function in this line is begin(), and that the potential danger occurs after begin() has already returned. That means that there is no way for vector<> to protect against it with a try block. The only way to ensure that it doesn't happen is to impose another requirement on the type T. No operation on T, including assignment, may put an object of type T into an inconsistent state.

```
1: reverse_iterator rbegin() { return reverse_iterator(end()); }
2: reverse_iterator rend() { return reverse_iterator(begin()); }
```

No new issues. Reverse_iterator's constructor simply invokes the pointer's copy constructor.

ASSIGNMENT OPERATOR

```
 1: template <class T, class Alloc>
 2: vector<T, Alloc>& vector<T, Alloc>::operator=(const vector<T, Alloc>& x) {
 3:   if (&x == this) return *this;
 4:   if (x.size() > capacity()) {
 5:     destroy(start, finish);
 6:     deallocate();
 7:     start = data_allocator::allocate(x.end() - x.begin());
 8:     end_of_storage = uninitialized_copy(x.begin(), x.end(), start);
 9:   } else if (size() >= x.size()) {
10:     vector<T, Alloc>::iterator i = copy(x.begin(), x.end(), begin());
11:     destroy(i, finish);
12:   } else {
13:     copy(x.begin(), x.begin() + size(), begin());
14:     uninitialized_copy(x.begin() + size(), x.end(), begin() + size());
15:   }
16:   finish = begin() + x.size();
17:   return *this;
18: }
```

The clause from lines 4 through 8 raises no new issues. The operations in lines 7 and 8 can potentially fail. If line 7 fails, it's necessary to assign values to start, finish, and end_of_storage such that the vector's destructor will succeed. (The two obvious choices are null pointers, or, by exchanging lines 7-8 and 5-6, the old contents of the vector.) If line 8 fails, it's necessary to do that and also to deallocate the storage that was allocated in line 7. This is all straightforward.

The clause in lines 10 and 11 also raises no new issues. Copy does nothing but iterator operations (in this case, pointer operations) and assignment. We already know that pointer operations are not permitted to throw exceptions. Any of the assignments in copy() might throw exceptions, but that's harmless. (It's harmless because we already know that an assignment operation involving objects of type T may not leave either of the objects in an inconsistent state.) In the event of a failure in line 10, finish should not be updated.

Clause in lines 13 and 14 also raises no new issues. If anything in line 13 or 14 throws an exception, finish should not be updated.

SWAP

```
1: void swap(vector<T, Alloc>& x) {
2:   ::swap(start, x.start);
3:   ::swap(finish, x.finish);
4:   ::swap(end_of_storage, x.end_of_storage);
5: }
```

No new issues, but a very clean proof (if any proof is still needed) that user-defined pointer types should not be permitted to throw exceptions upon assignment. Suppose that V1's members (start, finish, end_of_storage) are initially (s1, f1, e1), and V2's are initially (s2, f2, e2). Suppose that line 2 succeeds, and that the first pointer assignment in line 3 succeeds, and the second fails. In that case, V1 will be left in the state (s2, f2, e1) and V2 in the state (s1, f2, e2). The value f1 has been lost, and there is no way to recover it. Without it, there is no way that V2 can be destroyed.

RESERVE

```
 1: void reserve(size_type n) {
 2:   if (capacity() < n) {
 3:     const size_type old_size = size();
 4:     const iterator tmp = data_allocator::allocate(n);
 5:     uninitialized_copy(begin(), end(), tmp);
 6:     destroy(start, finish);
 7:     deallocate();
 8:     start = tmp;
 9:     finish = tmp + old_size;
10:     end_of_storage = start + n;
11:   }
12: }
```

No new issues. The operations in lines 2-3 and 6-10 can't fail. The operations in lines 4-5 can; a failure in line 4 requires no cleanup work, and a failure in line 5 merely leaves us with a block of raw storage that must be deallocated.

INSERTION

```
1: void push_back(const T& x) {
2:   if (finish != end_of_storage) {
3:     construct(finish, x);
4:     ++finish;
5:   } else
6:     insert_aux(end(), x);
7: }
```

The clause in lines 3 and 4 raise no new issues. Line 3 could throw an exception, but this does not affect the validity of the vector. Line 4 is yet another example of why it would be very bad to permit pointers to throw exceptions.

Lines 6 does nothing but call insert_aux, so we'll look at that separately.

```
1: iterator insert(iterator position, const T& x) {
2:   size_type n = position - begin();
3:   if (finish != end_of_storage && position == end()) {
4:     construct(finish, x);
5:     ++finish;
```

```
6:   } else
7:     insert_aux(position, x);
8:   return begin() + n;
9: }
```

Line 2 can't throw an exception. Lines 4 and 5 are equivalent to push_back, which we discussed above. Line 7 simply calls insert_aux, which we've already postponed. Line 8 can't throw an exception, and will never be executed if anything else throws an exception.

```
 1: template <class T, class Alloc>
 2: void vector<T, Alloc>::insert(iterator position, size_type n, const T& x) {
 3:   if (n == 0) return;
 4:   if (end_of_storage - finish >= n) {
 5:     if (end() - position > n) {
 6:       uninitialized_copy(end() - n, end(), end());
 7:       copy_backward(position, end() - n, end());
 8:       fill(position, position + n, x);
 9:     } else {
10:       uninitialized_copy(position, end(), position + n);
11:       fill(position, end(), x);
12:       uninitialized_fill_n(end(), n - (end() - position), x);
13:     }
14:     finish += n;
15:   } else {
16:     const size_type old_size = size();
17:     const size_type len = old_size + max(old_size, n);
18:     const iterator tmp = data_allocator::allocate(len);
19:     uninitialized_copy(begin(), position, tmp);
20:     uninitialized_fill_n(tmp + (position - begin()), n, x);
21:     uninitialized_copy(position, end(), tmp + (position - begin() + n));
22:     destroy(begin(), end());
23:     deallocate();
24:     end_of_storage = tmp + len;
25:     finish = tmp + old_size + n;
26:     start = tmp;
27:   }
28: }
```

The first clause (n == 0) is trivial.

The second clause (no reallocation is necessary, and all insertions go into previously initialized elements) presents no problems. It is harmless for line 6 to throw an exception, provided that uninitialized_copy adheres to the cleanup requirement that we've already discussed. It is almost harmless for lines 7 or 8 to throw an exception: if T adheres to the requirement that "t1 = t2" never puts t1 into an inconsistent state, then no vector elements will be corrupted. The problem is that if there is an exception in lines 7 or 8, then line 6 will have created initialized elements that are outside the range [start, finish). Either we have to have a try block here to destroy those elements, or else we have to move the "finish += n" immediately after line 6.

The third clause (no reallocation is necesary, and some insertions go into uninitialized elements) is similar to the second, but slightly more complicated. Again, it is harmless for line 10 to throw an exception. If line 11 or 12 throws an exception, then, again, there will be initialized elements outside [start, finish). The simplest way to deal with this problem is to increment finish once after line 10, and a second time after line 12.

The fourth clause (reallocation is necessary) is reasonable simple.

It is harmless for line 18 to throw an exception. If any of lines 19 through 21 throw an exception, then we must (1) destroy any elements that were created in the previous lines; and (2) deallocate the storage allocated in line 18. None of the lines from 22 through 26 can throw exceptions.

```
 1: template <class T, class Alloc>
 2: void vector<T, Alloc>::insert(iterator position,
 3:                               const_iterator first,
 4:                               const_iterator last) {
 5:   if (first == last) return;
 6:   size_type n = 0;
 7:   distance(first, last, n);
 8:   if (end_of_storage - finish >= n) {
 9:     if (end() - position > n) {
10:       uninitialized_copy(end() - n, end(), end());
11:       copy_backward(position, end() - n, end());
12:       copy(first, last, position);
13:     } else {
14:       uninitialized_copy(position, end(), position + n);
15:       copy(first, first + (end() - position), position);
16:       uninitialized_copy(first + (end() - position), last, end());
17:     }
18:     finish += n;
19:   } else {
20:     const size_type old_size = size();
21:     const size_type len = old_size + max(old_size, n);
22:     const iterator tmp = data_allocator::allocate(len);
23:     uninitialized_copy(begin(), position, tmp);
24:     uninitialized_copy(first, last, tmp + (position - begin()));
25:     uninitialized_copy(position, end(), tmp + (position - begin() + n));
26:     destroy(begin(), end());
27:     deallocate();
28:     end_of_storage = tmp + len;
29:     finish = tmp + old_size + n;
30:     start = tmp;
31:   }
32: }
```

This version of insert raises no new issues. The only difference is that it uses copy/uninitialized_copy instead of fill/uninitialized_fill, and the analysis is identical.

```
 1: template <class T, class Alloc>
 2: void vector<T, Alloc>::insert_aux(iterator position, const T& x) {
 3:   if (finish != end_of_storage) {
 4:     construct(finish, *(finish - 1));
 5:     T x_copy = x;
 6:     copy_backward(position, finish - 1, finish);
 7:     *position = x_copy;
 8:     ++finish;
 9:   } else {
10:     const size_type old_size = size();
11:     const size_type len = old_size != 0 ? 2 * old_size : 1;
12:     const iterator tmp = data_allocator::allocate(len);
13:     uninitialized_copy(begin(), position, tmp);
14:     construct(tmp + (position - begin()), x);
15:     uninitialized_copy(position, end(), tmp + (position - begin()) + 1);
16:     destroy(begin(), end());
17:     deallocate();
18:     end_of_storage = tmp + len;
19:     finish = tmp + old_size + 1;
20:     start = tmp;
21:   }
```

```
22: }
```

This is also essentially the same as the previous version of insert, except simpler. In the clause in lines 4 through 8, all we have to make sure of is that line 8 is moved immediately after line 4. After that, it's harmless for any of the next lines to throw exceptions.

Similarly, in the second clause, we only have to worry about lines 13 through 15. Again, all we have to do is make sure that (1) if any of them throw exceptions, we deallocate the storage allocated in line 12; and (2) if line 14 or 15 throws an exception, we destroy the elements constructed in line 13 and/or 14.

ERASURE

```
 1: void pop_back() {
 2:   --finish;
 3:   destroy(finish);
 4: }
 5: void erase(iterator position) {
 6:   if (position + 1 != end())
 7:     copy(position + 1, end(), position);
 8:   --finish;
 9:   destroy(finish);
10: }
11: void erase(iterator first, iterator last) {
12:   vector<T, Alloc>::iterator i = copy(last, end(), first);
13:   destroy(i, finish);
14:   finish = finish - (last - first);
15: }
16: void clear() { erase(begin(), end()); }
```

No problems. The only operations that can throw exceptions are the ones in lines 7 and 12, and in neither place could an exception cause

RESIZE

```
1: void resize(size_type new_size, const T& x) {
2:   if (new_size < size())
3:     erase(begin() + new_size, end());
4:   else
5:     insert(end(), new_size - size(), x);
6: }
7: void resize(size_type new_size) { resize(new_size, T()); }
```

No new issues. This is nothing but insert() or erase(), and both have already been dealt with.

UNINITIALIZED_* algorithms.

We need a strong guarantee: if any operation in them throws an exception, then previously constructed elements will be destroyed. Fortunately, this guarantee is possible.

```
 1: template <class InputIterator, class ForwardIterator>
 2: ForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
 3:                                    ForwardIterator result) {
 4:     while (first != last) construct(&*result++, *first++);
 5:     return result;
 6: }
 7:
 8: template <class ForwardIterator, class T>
```

```
 9: void uninitialized_fill(ForwardIterator first, ForwardIterator last,
10:                        const T& x) {
11:     while (first != last) construct(&*first++, x);
12: }
13:
14: template <class ForwardIterator, class Size, class T>
15: ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n,
16:                                      const T& x) {
17:     while (n--) construct(&*first++, x);
18:     return first;
19: }
```

The only operations here are iterator arithmetic, iterator dereference, and construction. Construction can certainly fail. So what we need to do is keep track of the range of fully constructed elements. (Initially the range is [first, first); after successful completion, it is [first, last).) If any of the constructors fail, we call destroy() on the range of constructed elements.

Note that this method assumes that destroy() can't possibly fail. Since destroy() also uses iterator arithmetic and dereference, this means that the uninitialized_* algorithms can provide the necessary guarantee iff we impose a restriction on their arguments: incrementing or dereferencing a valid iterator may not throw an exception. (Note that this restriction applies only to these three algorithms, not to the entire library.) Also, as usual, the destructor can't throw an exception.

In the case of vector<>, we already have that restriction. The iterators are pointers, and, for other reasons, we already know that valid pointer arithmetic may not result in exceptions.