Doc. No.: J16/97-0048R1 = WG21/N1086R1
Date: 4 July 1997
Project: Programming Language C++
Reply to: Dave Abrahams <abrahams@motu.com>
Greg Colvin <greg@imrgold.com>

Making the C++ Standard Library Exception Safe

The current working paper specifies that if client code used to instantiate a standard library template throws an exception, the behavior is undefined. As a consequence it is impossible to write portable code that uses exceptions and the standard library without invoking undefined behavior. It is even impossible to use many of the standard library components together.

This proposal recommends changes that allow people to write useful, portable code using exceptions with the standard library. These changes amount to a promise that if the user of the standard library meets certain requirements, then the standard library makes certain guarantees:

1. The "basic guarantee" that the standard library will not leak memory or constructed objects
2. The "strong guarantee" that in some cases if a function throws an exception, that function will have no effects
3. The guarantee that in some cases a function will not throw an exception.

This proposal differs from J16/97-0048=WG21/N1086 by simplified changes to 17.3.4.8, corrected reference code for 20.4.4, added requirements on *list::sort()*, removed requirements on allocator copy and assignment, and minor editorial fixes.

We believe the changes we propose to Chapters 17 and 20, which provide the basic guarantee for all library components, now have substantially the same effect as the corresponding changes proposed by Matt Austern in J16/97-0039=N1077. We also remain convinced that the changes we propose for Chapter 23, which provide guarantees beyond the basic guarantee for some important container functions, are necessary for effective use of the standard library containers.

Example #1

Consider this simple class template as a test case:

```
template <class T>
class SearchableStack
{
public:
   void push(const T& t);           // Adds to set_impl and adds the
                                    // iterator to the end of list_impl
   void pop();                      // Pops the most-recently pushed item
   bool contains(const T& t) const; // true iff the stack contains t
   bool empty() const;
   const T& top() const;
   void swap(SearchableStack<T>&);
private:
   set<T> set_impl;
   list<set<T>::iterator> list_impl;
};
```

We need to maintain the following invariant:

> For each dereferenceable iterator value *s* in *set_impl* there is exactly one dereferenceable iterator value *l* of *list_impl* such that $*l = s$. *list_impl* contains only dereferenceable iterator values in *s*.

Consider this naive implementation of the push() member function:

```
template <class T>
void SearchableStack<T>::push(const T& t)
{
   set<T>::iterator i = set_impl.insert(t);
   list_impl.push_back(i);
}
```

What's wrong with this function? The most obvious problem is that if the second line causes an exception, our invariant has been broken and our class is corrupted. To recover from this exception, we need something more like this:

```
template <class T>                            // 1
void SearchableStack<T>::push(const T& t)     // 2
{                                             // 3
   set<T>::iterator i = set_impl.insert(t);   // 4
   try                                        // 5
   {                                          // 6
      list_impl.push_back(i);                 // 7
   }                                          // 8
   catch(...)                                 // 9
   {                                          // 10
      set_impl.erase(i);                      // 11
      throw;                                  // 12
   }                                          // 13
}                                             // 14
```

If we want this code to work, what requirements on the library are implied?

First of all, line 7 must satisfy the strong guarantee: if an exception is thrown it must have no effects. If the list is allowed to change in arbitrary ways when an exception is thrown our invariant will be broken.

Line 4 must also satisfy the strong guarantee by a similar argument. It's a little trickier to justify because one needs to show that there's no other way to recover from *set<T>::insert(const T&)* using a *try/catch* block. If you don't consider emptying the *SearchableStack* a suitable recovery action, though, it's easy to see that inserting a single element must be "strong".

Lastly, line 11 must not throw an exception or our program will terminate. We could wrap the erase in a *try/catch* block to prevent termination, but our list would be in an unknown state (was the element erased or not?), which begs the question of preserving the invariant. So the following functions may not throw exceptions:

```
set<T>::erase(set<T>::iterator)
set<T>::iterator::iterator(const set<T>::iterator&)
```

Example #2

We've seen why it can be important to have the strong guarantee for some operations, but many of the most useful parts of the library are algorithms like *sort()* which operate on iterator ranges, and in general it isn't possible to roll back changes to an iterator range.

Fortunately these operations can often be strengthened when used on containers:

```
template <class T, class Allocator>          // 1
void StrongSort(vector<T,A>& v)              // 2
{                                            // 3
   vector<T,Allocator> tmp(c);               // 4
   sort(tmp.begin(), tmp.end());             // 5
   v.swap(tmp);                              // 6
}                                            // 7
```

This technique works as long as line 6 gives the strong guarantee.

Example #3:

What about the *swap()* member function on *SearchableStack*? Here's the implementation:

```
template <class T>
void SearchableStack<T>::swap(SearchableStack<T>& other)
{
  set_impl.swap(other.set_impl);
  list_impl.swap(other.list_impl);
}
```

To preserve the invariant, one of the two *swap()* functions must give the strong guarantee, and one must succeed unconditionally. For the standard sequences we guarantee that *swap()* will succeed without preconditions, but for the standard associative containers we require that their *Compare* object must be assignable and copy-constructable without throwing an exception, which is already true of the default *Compare* object.

Lessons from the Examples

Insertion in containers usually claims new resources, and thus can easily cause an exception. In order to allow the user to preserve invariants, it is important that the operations used to recover from insertion be required not to throw whenever possible. This affects the following standard container functions:

- *erase()* for *list*, *map*, *set*, *multimap*, and *multiset*.
- *pop_back()* for *deque*, *list*, and *vector*
- *pop_front()* for *deque* and *list*
- copy constructors and assignment operators for all standard container iterators

Additionally, it is important to know when single-element insertions provide the strong guarantee. Recovery from a multiple-element insertion is only possible if there can be some assurance of the container's contents after each single insertion. The following standard container functions should give the strong guarantee:

- single-element insertions for *list*, *map*, *set*, *multimap*, and *multiset*
- *push_back()* for *deque*, *list*, and *vector*
- *push_front()* for *deque* and *list*

[Except for *deque* and *vector*, we believe that allowing client exceptions may already imply the strong guarantee for single element insert and push operations because of the existing guarantees that iterators and references remain valid.]

Finally, some standard library functions should be required to exhibit useful exception behavior, at least in some circumstances (see the proposed changes for details):

- all destructors
- *swap()* for all containers
- *insert()* and *erase()* for *deque* and *vector*
- multiple element *insert()*, *splice()*, *remove()*, *unique()*, *merge()*, and *sort()* for *list*

Working Paper Changes: Chapter 17

Chapter 17 currently prohibits client code from throwing exceptions, and allows most library code to throw exceptions at any time. We propose to fix that.

To allow client code to throw exceptions, replace the last two sentences of 17.3.3.6, paragraph 2, with the following:

- for types used as template arguments when instantiating a template component, if the operations on the type do not implement the semantics of the applicable Requirements subclause (20.1.5, 23.1, 24.1, 26.1). Operations on such types can report a failure by throwing an exception unless otherwise specified.
- if any replacement function or handler function or destructor operation throws an exception, unless specifically allowed in the applicable Required behavior paragraph.

To better specify when the library may throw exceptions, replace paragraph 3 of section 17.3.4.8 with the following:

> No destructor operation defined in the C++ Standard library will throw an exception. Any other functions defined in the C++ Standard library that do not have an exception-specification may throw implementation-defined exceptions. An implementation may strengthen this implicit exception-specification by adding an explicit one.

Working Paper Changes: Chapter 20

The allocator requirements in 20.1.5 currently specify that for an allocator class *X* on type *T* the *X::pointer* and *X::const_pointer* types are *T** and *const T**, respectively, but encourage more general types. When we generalize these types we must specify that valid operations on generalized pointers will not throw. To ensure that container destructors do not throw we must require that *X::deallocate()* does not throw, by adding "must not throw exceptions" to the notes in Table 32 20.1.5, and adding a *throw()* specification to *allocator::deallocate()* in 20.4.1.

Chapter 20 specifies the effects of *uninitialized_copy()*, *uninitialized_fill()* and *uninitialized_fill_n()* with reference C++ implementations. We propose to change those implementations to meet the strong guarantee.

Replace the Effects paragraph of 20.4.4.1 (*uninitialized_copy*) with:

Effects:

```
typedef typename iterator_traits<ForwardIterator>::value_type T;
ForwardIterator save = result;
try {
   for (; first != last; ++result, ++first)
      new (static_cast<void*>(&*result)) T(*first);
} catch(...) {
   for (; save != result; ++save)
      save->~T();
   throw;
}
```

Replace the Effects paragraph of 20.4.4.2 (*uninitialized_fill*) with:

Effects:

```
typedef typename iterator_traits<ForwardIterator>::value_type T;
ForwardIterator save = first;
try {
   for (; first != last; ++first)
      new (static_cast<void*>(&*first)) T(x);
} catch(...) {
   for (; save != first; ++save)
      save->~T();
   throw;
}
```

Replace the Effects paragraph of 20.4.4.3 (*uninitialized_fill_n*) with:

Effects:

```
typedef typename iterator_traits<ForwardIterator>::value_type T;
ForwardIterator save = first;
try {
   for (; n--; ++first)
      new (static_cast<void*>(&*first)) T(x);
} catch(...) {
    for (; save != first; ++save)
       save->~T();
    throw;
}
```

Working Paper Changes: Chapter 23

The changes above suffice to ensure that the Standard library makes the basic guarantee of not leaking resources. Further guarantees require some additions to Chapter 23.

Append the following paragraph to section 23.1 (Container requirements)

> Unless otherwise specified all container types defined in this clause meet the following additional requirements:
>
> - if an exception is thrown by an **`insert()`** function while inserting a single element that function has no effects.
> - if an exception is thrown by a **`push_back()`** or **`pop_front()`** function that function has no effects.
> - no **`erase()`**, **`pop_back()`** or **`pop_front()`** function throws an exception.
> - no copy constructor or assignment operator of a returned iterator throws an exception.
> - no **`swap()`** function throws an exception unless that exception is thrown by the copy constructor or assignment operator of the container's **`Compare`** object (if any, see 23.1.2).

Insert the following paragraph after paragraph 1 of section 23.2.1.3 (*deque::insert*):

> Notes: If an exception is thrown other than by the copy constructor or assignment operator of **`T`** there are no effects.

Insert the following paragraph after paragraph 3 of section 23.2.1.3 (*deque::erase*):

> Throws: Nothing unless an exception is thrown by the copy constructor or assignment operator of **`T`**.

Append the following sentence to paragraph 1 of section 23.2.2.3 (*list::insert*):

> If an exception is thrown there are no effects.

Insert the following paragraph after paragraphs 4, 6 & 9 of section 23.2.2.4 (*list::splice*):

> Throws: Nothing.

Insert the following paragraph after paragraph 12 of section 23.2.2.4 (*list::remove* and *list::remove_if*):

> Throws: Nothing unless an exception is thrown by **`*i == value`** or **`pred(*i) != false`**.

Insert the following paragraph after paragraph 15 of section 23.2.2.4 (*list::unique*):

> Throws: Nothing unless an exception is thrown by **`*i == *(i-1)`** or **`pred(*i,*(i-1))`**.

Append the following sentence to paragraph 19 of section 23.2.2.4 (*list::merge*):

> If an exception is thrown other than by a comparison there are no effects.

Append the following sentence to paragraph 25 of section 23.2.2.4 (*list::sort*):

> If an exception is thrown the order of the elements in the list is indeterminate.

Append the following sentence to paragraph 1 of section 23.2.4.3 (*vector::insert*):

> If an exception is thrown other than by the copy constructor or assignment operator of **`T`** there are no effects.

Insert the following paragraph after paragraph 3 of section 23.2.4.3 (*vector::erase*):

> Throws: Nothing unless an exception is thrown by the copy constructor or assignment operator of **`T`**.

Generalized Pointer Types for Allocators J16/97-009R1 = WG21/N1047R1

Greg Colvin. Information Management Research.

This proposal differs from 97-0009=N1047 by requiring generalized pointers to convert to raw pointers, as was required before Kona; by requiring raw pointers to assign and convert to generalized pointers, so as to support null pointers and type conversions; and by incorporating those parts of Matt Austern's proposal (97-0040=N1078) which clarify pre-Kona requirements.

Changes to 20.1.5 [lib.allocator.requirements]

Change the table "Allocator requirements" as follows:

`X::pointer`	A mutable random access iterator, convertible to **`T*`** and **`X::const_pointer`**, whose value type, difference type, pointer type, reference type, and iterator category are, respectively, **`X::value_type`**, **`X::difference_type`**, **`X::value_type*`**, **`X::reference`**, and **`random_access_iterator_tag`**. Valid operations on **`X::pointer`** must not throw exceptions.	
`X::const_pointer`	A constant random access iterator, convertible to **`const T*`**, whose value type, difference type, pointer type, reference type, and iterator category are, respectively, **`X::value_type`**, **`X::difference_type`**, **`const X::value_type*`**, **`X::const_reference`**, and **`random_access_iterator_tag`**. Valid operations on **`X::const_pointer`** must not throw exceptions.	
`X::size_type`	An integral type that can represent the size of the largest object in the allocation model, and that can represent every non-negative value of **`X::difference_type`**. **`X::size_type`** and **`Y::size_type`** are the same types.	
`X::difference_type`	An integral type that can represent the difference between any two pointers in the allocation model. **`X::difference_type`** and **`Y::difference_type`** are the same types.	
`a.allocate(n)` **`a.allocate(n,u)`**	**`X::pointer`**	Memory is allocated for **`n`** objects of type **`T`** but objects are not constructed. **`allocate`** may throw an appropriate exception. If the return value is denoted **`p`**, then **`p+n`** is a past-the-end iterator and all of the pointers in the range **`[p,p+n)`** are dereferenceable iterators. 204)
a.**`deallocate(p,n)`**	(not used)	All **`n`** **`T`** objects in the area pointed to by **`p`** must be destroyed prior to this call. **`n`** must match the value passed to **`allocate`** to obtain this memory. **`p`** must not be null. **`deallocate`** must not throw exceptions.
`X::pointer x(&r);`		Post: x **`== p`**
`X::pointer x; x = &r;`		Post: x **`== p`**
`X::const_pointer x(&s);` Post: x **`== q`**		
`X::const_pointer x; x = &s;`		Post: x **`== q`**

Replace paragraphs 4 and 5 with the following sentence:

> The semantics of containers and algorithms when allocator instances compare non-equal are implementation defined.

Changes to 21.3.1 [lib.string.cons]

Append the following sentence to paragraph 1:

> The **`Allocator`** argument must meet the further requirement that the typedef members **`pointer`**, **`const_pointer`**, **`size_type`**, and **`difference_type`** be **`charT*`**, **`const charT*`**, **`size_t`**, and **`ptrdiff_t`**, respectively.

```
///////////////////////////////////////////////////////////////////////////
// ALLOC.H  Custom allocator for testing Microsoft C++ 5.0 STL implemention.
// Copyright 1997 Gregory Colvin.  May be distributed free with this notice.

#include <cstddef>
#include <iterator>

template<typename T> class Alloc {

   // minimal base class for pointer implementations
   template<typename P,typename U> class base_ptr
   : public std::iterator<std::random_access_iterator_tag,U,ptrdiff_t> {
      U* ptr;
   public:
      base_ptr(U* p) : ptr(p) {}

      operator U*  () const { return ptr; }
      U* operator->() const { return ptr; }

      U&       operator[](size_t i)       { return ptr[i]; }
      const U& operator[](size_t i) const { return ptr[i]; }

      P& operator= (U* p)        { ptr =  p; return static_cast<P&>(*this); }
      P& operator+=(ptrdiff_t n) { ptr += n; return static_cast<P&>(*this); }
      P& operator-=(ptrdiff_t n) { ptr -= n; return static_cast<P&>(*this); }
      P& operator++()            { ++ptr;    return static_cast<P&>(*this); }
      P& operator--()            { --ptr;    return static_cast<P&>(*this); }
      P  operator++(int)         { P p(ptr); ++ptr; return p; }
      P  operator--(int)         { P p(ptr); --ptr; return p; }

      // compiler insisted on both ptrdiff_t and size_t versions
      P operator+(ptrdiff_t n) const { return P(ptr+n); }
      P operator-(ptrdiff_t n) const { return P(ptr-n); }
      P operator+(size_t n)    const { return P(ptr+n); }
      P operator-(size_t n)    const { return P(ptr-n); }

      // compiler could handle only == and != by conversion to T*
      bool operator< (const T* p) const { return ptr <  p; }
      bool operator<=(const T* p) const { return ptr <= p; }
      bool operator>=(const T* p) const { return ptr >= p; }
      bool operator> (const T* p) const { return ptr >  p; }
   };

public:

   // types
   struct pointer : base_ptr<pointer,T> {
      pointer(T* p=0) : base_ptr<pointer,T>(p) {}
   };
   struct const_pointer : base_ptr<const_pointer,const T> {
      const_pointer(const T* p=0)      : base_ptr<const_pointer,const T>(p) {}
      const_pointer(const pointer& p) : base_ptr<const_pointer,const T>(p) {}
   };
   typedef T& reference;
   typedef const T& const_reference;
   typedef T value_type;
   typedef size_t size_type;
   typedef ptrdiff_t difference_type;

   // functions
   pointer allocate(size_t n,const void*) {
      return (T*)operator new(n*sizeof T);
   }
   void deallocate(pointer p, size_t) {
      operator delete(p);
   }
   void construct(pointer p, const T& t) {
      new((void *)p) T(t);
   }
   void destroy(pointer p) {
      p->~T();
   }
   size_t max_size() const {
      size_t n = (size_t)(-1) / sizeof(T);
      return 0 < n ? n : 1;
```

```
   }

   // redundancy
   pointer address(T& t) { return pointer(&t); }
   const_pointer address(const T& t) { return const_pointer(&t); }

   // uselessness
   bool operator==(const Alloc<T>&) const { return true; }
   bool operator!=(const Alloc<T>&) const { return false; }

   // compromise
   #ifdef _MSC_VER
      char* _Charalloc(size_t n) { return (char*)operator new(n*sizeof T); }
   #else
      template<typename U> struct rebind { typedef allocator<U> other; };
   #endif
};
```

```
/////////////////////////////////////////////////////////////////////////////
// ALLOC.CPP  Simple test program for custom allocator.
// Copyright 1997 Gregory Colvin.  May be distributed free with this notice.
#include "alloc.h"

#include <iostream>
#include <algorithm>
#include <string>
#include <deque>
#include <vector>
using namespace std;

template<class Container> void print(const Container& c)
{
   for (Container::const_iterator p=c.begin(), q=c.end(); p!=q; ++p)
      cout << *p << ' ';
   cout << endl;
}

template<class Container> void test(int argc, char** argv)
{
   Container args;
   while (--argc > 0)
      args.push_back(*++argv);
   print(args);
   make_heap(args.begin(),args.end());
   print(args);
   sort_heap(args.begin(),args.end());
   print(args);
   random_shuffle(args.begin(),args.end());
   print(args);
   sort(args.begin(),args.end());
   print(args);
   random_shuffle(args.begin(),args.end());
   print(args);
   stable_sort(args.begin(),args.end());
   print(args);
}

main(int argc, char** argv)
{
   test<deque<string,Alloc<string> > >(argc,argv);
   test<vector<string,Alloc<string> > >(argc,argv);
}
```