# STL Exception Handling Contract

*Dave Abrahams* <abrahams@motu.com>

## Introduction

This documents the exception-handling behavior of the Standard Template Library available at <http://www.ipmce.su/~fbp/stl/>, which is an adaptation of the work done at Silicon Graphics by Matt Austern. Although this document is not intended as a proposal in itself, I hope that it illustrates the intent of a proposal that Greg Colvin and I plan to submit for the London meeting. Our proposal will be substantially similar in effect, but because of the realities of the way the standard is written we expect this document to be more explicit and easier to understand.

---

## Basic Library Guarantees

**This version of the STL makes the guarantee that no resources are leaked in the face of exceptions**.

In particular, this means:

- By the time a container's destructor completes:
    - It has returned all memory it has allocated to the appropriate deallocation function.
    - The destructor has been called for all objects constructed by the container.
- Algorithms destroy all temporary objects and deallocate all temporary memory even if the algorithm does not complete due to an exception.
- Algorithms which construct objects (e.g. `uninitialized_fill`) either complete successfully or destroy any objects they have constructed at the time of the exception.
- Algorithms which destroy objects always succeed.

**Additionally:**

- Algorithms which operate on ranges of objects leave only fully-constructed objects in those ranges if they terminate due to an exception.
- Containers continue to fulfill all of their requirements, even after an exception occurs during a mutating function. For example, a map will never give an inaccurate report of its size, or fail to meet its performance requirements because the tree that implements it has become unbalanced.
- A stronger guarantee is available for some operations: that *if the operation terminates due to an exception, program state will remain unchanged* . For example, `vector<T,A>::push_back()` leaves the vector unchanged if an exception is thrown, provided the library client fulfills the basic requirements below. For some operations, the "strong guarantee" is available if additional requirements are filled.

---

## Basic Client Requirements

**The library guarantees above are conditional on some requirements that library clients must fulfill.**

**The following operations must return normally - they are forbidden to terminate due to an exception:**

- Destructors of any classes used by the library. This includes all classes used as library template parameters. It also includes all classes which fulfill "type requirements" of classes used as library templates- an allocator's `size_type`, for example.
- Valid uses of any of the required functionality of the following types. Note that invalid uses (e.g. comparison of two iterators from different containers) are not prohibited from throwing an exception. Presumably, invalid uses would cause worse problems than resource leaks:
  - The ForwardIterator arguments to the following:
    - `uninitialized_copy(InputIterator first, InputIterator last, ForwardIterator result)`
    - `uninitialized_fill(ForwardIterator first, ForwardIterator last, const T& x)`
    - `uninitialized_fill_n(ForwardIterator first, Size n, const T& x)`
    - `destroy(ForwardIterator first, ForwardIterator last)`
  - An allocator's `deallocate()` function
  - Any of the required allocator types:
    - `pointer`
    - `const_pointer`
    - `reference`
    - `const_reference`
    - `size_type`
    - `difference_type`

---

Note: Algorithms like `copy()` expect that they are copying into real objects. The use of `raw_storage_iterator` with most algorithms is inherently exception-unsafe:

```
// objects of the same type as *iterator1 may be leaked if a failure occurs.
copy( iterator1, iterator2, raw_storage_iterator( ptr ) );
```

Furthermore, there is no way to properly recover from this using an enclosing `try`/`catch` block, because `raw_storage_iterator` has no function in its public interface to tell you how far it has been advanced.

---

# The "Strong Guarantee"

In most programs, many objects will be destroyed automatically during exception-unwinding. For these, the basic guarantee that resources won't be leaked is good enough. If a program hopes to survive an exception and continue running, though, it probably also uses long-lived containers which are expected to survive past exception-recovery in a known state. For example, a program could maintain a list of objects representing tasks it is working on. If adding a task to that list fails, the program may still need to rely on the list. If the list must survive an exception intact, we need the strong guarantee. You can get the strong guarantee by "brute force" for any container operation as follows, provided the container's `swap()` member function can't fail (this is true for most real-world containers):

```
container_type container_copy( original_container );
container_copy.mutating_operation(...);
original_container.swap( container_copy );
```

Fortunately, many mutating operations give the strong guarantee with no additional requirements on the client. To get the strong guarantee for others, you can either use the above technique or conform to some additional requirements.

# Operations that give the "strong guarantee" with no additional requirements

*( Operations labelled with * are guaranteed to return normallyif all basic requirements have been met)*

- uninitialized_fill()
- uninitialized_copy()
- uninitialized_fill_n()
- deque<T,A> member functions:
  - swap(deque<T,A>&) *
  - push_back(const T&)
  - pop_back() *
  - push_front(const T&)
  - pop_front() *
- list<T,A> member functions:
  - insert(iterator position, const T& x = T())
  - insert(iterator position)
  - push_back(const T&)
  - pop_back() *
  - push_front(const T&)
  - pop_front() *
  - splice(iterator position, list<T,Allocator>& x) *
  - splice(iterator position, list<T,Allocator>& x, iterator i) *
  - splice(iterator position, list<T,Allocator>& x, iterator first, iterator last) *
  - swap(list<T,A>&) *
  - reverse() *
  - erase(iterator position) *
  - erase(iterator first, iterator last) *
- vector<T, A> member functions:
  - reserve(size_type n)
  - swap(vector<T,A>&) *
  - push_back(const T&)
  - pop_back() *
- bit_vector<A> member functions:
  - reserve(size_type n)
  - swap(bit_vector&) *
  - push_back(const T&)
  - pop_back() *
  - insert(iterator position, bool x = bool())
  - insert(iterator position)
  - insert(iterator position, const_iterator first, const_iterator last)
  - insert(iterator position, const bool* first, const bool* last)
  - insert(iterator position, size_type n, bool x)
  - erase(iterator position) *
  - erase(iterator first, iterator last) *
- map<K, T, C, A> member functions:
  - operator[](const key_type& k)
  - insert(iterator position, const value_type& x)
  - insert(const value_type& x)
  - erase(const key_type& x) *
  - erase(iterator position) *

- erase(iterator first, iterator last) *

- set<K, C, A> member functions:
  - insert(iterator position, const value_type& x)
  - insert(const value_type& x)
  - erase(const key_type& x) *
  - erase(iterator position) *
  - erase(iterator first, iterator last) *
- multimap<K, T, C, A> member functions:
  - insert(iterator position, const value_type& x)
  - insert(const value_type& x)
  - erase(const key_type& x) *
  - erase(iterator position) *
  - erase(iterator first, iterator last) *
- multiset<K, C, A> member functions:
  - insert(iterator position, const value_type& x)
  - insert(const value_type& x)
  - erase(const key_type& x) *
  - erase(iterator position) *
  - erase(iterator first, iterator last) *
- hash_map<K, T, H, E, A> member functions:
  - insert_noresize(const value_type& obj)
  - erase(const key_type& key) *
  - erase(iterator position) *
  - erase(iterator first, iterator last) *
- hash_multimap<K, T, H, E, A> member functions:
  - insert_noresize(const value_type& obj)
  - erase(const key_type& key) *
  - erase(iterator position) *
  - erase(iterator first, iterator last) *
- hash_set<T, H, E, A> member functions:
  - insert_noresize(const value_type& obj)
  - erase(const key_type& key) *
  - erase(iterator position) *
  - erase(iterator first, iterator last) *
- hash_multiset<T, H, E, A> member functions:
  - insert_noresize(const value_type& obj)
  - erase(const key_type& key) *
  - erase(iterator position) *
  - erase(iterator first, iterator last) *
- clear() for all containers *
- all container const member functions *
- all constructors, by language definition (included for completeness)

---

# Strong guarantee requirements for other mutating container operations

### Definition of terms

| *Term* | *Meaning, when applied to a type* **T** (x *and* y *of type* T) |
|---|---|
| "guaranteed copyable" | T z(x) and x = y may not exit via exception. |
| "guaranteed equality-comparable" | x == y may not exit via exception. |
| "guaranteed comparable" | x < y may not exit via exception. |

---

### deque<T,A> member functions

<table>
<tr><th>Function</th><th>Requirements</th></tr>
<tr><td>insert(iterator position, const T& x)<br>insert(iterator position)</td><td>position == begin() || position == end()<br>OR T guaranteed copyable</td></tr>
<tr><td>erase(iterator position)</td><td>position == begin() || position == end() - 1<br>OR T guaranteed copyable</td></tr>
<tr><td>erase(iterator first, iterator last)</td><td>first == begin() || last == end()<br>OR T guaranteed copyable</td></tr>
<tr><td>resize(size_type new_size, const T& x)<br>resize(size_type new_size)</td><td>new_size == size() + 1<br>OR new_size <= size() *</td></tr>
</table>

---

### list<T,A> member functions

| *Function* | *Requirements* |
|---|---|
| remove(const T& value) | T guaranteed equality-comparable |
| unique() | T guaranteed equality-comparable |
| merge(list<T, Alloc>& x) | T guaranteed comparable |
| sort() | T guaranteed comparable |

---

### vector<T,A> member functions

| *Function* | *Requirements* |
| --- | --- |
| `insert(iterator position, const T& x)`<br>`insert(iterator position)` | `position == end()`<br>**OR** `T` guaranteed copyable |
| `insert (iterator position, const_iterator first, const_iterator last);`<br><br>`void insert (iterator position, size_type n, const T& x);` | `T` guaranteed copyable |
| `erase(iterator position)` | `position == end() - 1`<br>**OR** `T` guaranteed copyable |
| `erase(iterator first, iterator last)` | `last == end()`<br>**OR** `T` guaranteed copyable |
| `resize(size_type new_size, const T& x)`<br>`resize(size_type new_size)` | `new_size == size() + 1`<br>**OR** `T` guaranteed copyable<br>**OR** `new_size <= size()` * |

---

## Basic Associative Container member functions

| *Function* | *Requirements* |
| --- | --- |
| `map<Key,T,Compare,A>::swap(map<Key,T,Compare,A>& amp;)` | `Compare` guaranteed copyable * |
| `multimap<Key,T,Compare,A>::swap(multimap<Key,T,Compare,A&g t;&)` | `Compare` guaranteed copyable * |
| `set<T,Compare,A>::swap(set<T,Compare,A>&)` | `Compare` guaranteed copyable * |
| `multiset<T,Compare,A>::swap(multiset<T,Compare,A>&)` | `Compare` guaranteed copyable * |

---

## hash_map<K, T, HashFcn, EqualKey, A> member functions

| *Function* | *Requirements* |
|---|---|
| swap(hash_map<K, T, HashFcn, EqualKey, A>&) | HashFcn and EqualKey guaranteed copyable * |
| insert(const value_type& obj) | bucket_count() >= size() + 1 |
| operator[](const key_type& k) | bucket_count() >= size() + 1 |

### hash_multimap<K, T, HashFcn, EqualKey, A> member functions

| *Function* | *Requirements* |
|---|---|
| swap(hash_multimap<K, T, HashFcn, EqualKey, A>&) | HashFcn and EqualKey guaranteed copyable * |
| insert(const value_type& obj) | bucket_count() >= size() + 1 |

### hash_set<T, HashFcn, EqualKey, A> member functions

| *Function* | *Requirements* |
|---|---|
| swap(hash_set<K, T, HashFcn, EqualKey, A>&) | HashFcn and EqualKey guaranteed copyable * |
| insert(const value_type& obj) | bucket_count() >= size() + 1 |

### hash_multiset<T, HashFcn, EqualKey, A> member functions

| *Function* | *Requirements* |
|---|---|
| swap(hash_multiset<K, T, HashFcn, EqualKey, A>&) | HashFcn and EqualKey guaranteed copyable * |
| insert(const value_type& obj) | bucket_count() >= size() + 1 |