
6.12. Errors and Exceptions inside the STL

Errors happen. They might be logical errors caused by the program (the programmer) or runtime errors caused by the context or the environment of a program (such as low memory). Both kinds of errors may be handled by exceptions. This section discusses how errors and exceptions are handled inside the STL.

6.12.1. Error Handling

The design goal of the STL was the best performance rather than the highest security. Error checking wastes time, so almost none is done. This is fine if you can program without making any errors but can be a catastrophe if you can't. Before the STL was adopted into the C++ standard library, discussions were held about whether to introduce more error checking. The majority decided not to, for two reasons:

1. Error checking reduces performance, and speed is still a general goal of programs. As mentioned, good performance was one of the design goals of the STL.
2. If you prefer safety over speed, you can still get it, either by adding wrappers or by using special versions of the STL. But when error checking is built into all basic operations, you can't program to avoid error checking to get better performance For example, when every subscript operation checks whether a range is valid, you can't write your own subscripts without checking. However, it is possible the other way around.

As a consequence, error checking is possible but usually not required inside the STL.

The C++ standard library states that any STL use that violates preconditions results in undefined behavior. Thus, if indexes, iterators, or ranges are not valid, the result is undefined. If you do not use a safe version of the STL, undefined memory access typically results, which causes some nasty side effects or even a crash. In this sense, the STL is as error prone as pointers are in C. Finding such errors could be very hard, especially without a safe version of the STL.

In particular, the use of the STL requires that the following be met:

- Iterators must be valid. For example, they must be initialized before they are used. Note that iterators may become invalid as a side effect of other operations. In particular, iterators become invalid
  - for vectors and deques, if elements are inserted or deleted or reallocation takes place, and
  - for unordered containers, if rehashing takes place (which also might be the result of an insertion).
- Iterators that refer to the past-the-end position have no element to which to refer. Thus, calling operator `*` or operator `->` is not allowed. This is especially true for the return values of the `end()`, `cend()`, and `rend()` container member functions.
- Ranges must be valid:
  - Both iterators that specify a range must refer to the same container.
  - The second iterator must be reachable from the first iterator.
- If more than one source range is used, the second and later ranges usually must have at least as many elements as the first one.
- Destination ranges must have enough elements that can be overwritten; otherwise, insert iterators must be used.

The following example shows some possible errors:

```
// stl/iterbug.cpp

#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> coll1;      // empty collection
    vector<int> coll2;      // empty collection

    // RUNTIME ERROR:
    // - beginning is behind the end of the range
    vector<int>::iterator pos = coll1.begin();
    reverse (++pos, coll1.end());

    // insert elements from 1 to 9 into coll1
    for (int i=1; i<=9; ++i) {
        coll1.push_back (i);
    }

    // RUNTIME ERROR:
    // - overwriting nonexisting elements
    copy (coll1.cbegin(), coll1.cend(),     // source
          coll2.begin());                   // destination
```

```
        // RUNTIME ERROR:
        // - collections mistaken
        // - cbegin() and cend() refer to different collections
        copy (coll1.cbegin(), coll2.cend(),     // source
              coll1.end());                     // destination
}
```

Note that because these errors occur at runtime, not at compile time, they cause undefined behavior.

There are many ways to make mistakes when using the STL, and the STL is not required to protect you from yourself. Thus, it is a good idea to use a "safe" STL, at least during software development. A first version of a safe STL was introduced by Cay Horstmann (see [*SafeSTL*]). Another example is the "STLport," which is available for free for almost any platform at [*STLport*]. In addition, library vendors now provide flags to enable a "safer" mode, which especially should be enabled during development.[16]

[16] For example, `gcc` provides the `-D_GLIBCXX_DEBUG` option for that.

6.12.2. Exception Handling

The STL almost never checks for logical errors. Therefore, almost no exceptions are generated by the STL itself owing to a logical problem. In fact, there are only two function calls for which the standard requires that it might cause an exception directly: the `at()` member function, which is the checked version of the subscript operator, and `reserve()` if the passed size of elements exceeds `max_size()`. Other than that, the standard requires that only the usual standard exceptions may occur, such as `bad_alloc` for lack of memory or exceptions of user-defined operations.

When are exceptions generated, and what happens to STL components when they are? For a long time during the standardization process of C++98, there was no defined behavior about this. In fact, every exception resulted in undefined behavior. Even the destruction of an STL container resulted in undefined behavior if an exception was thrown during one of its operations. Thus, the STL was useless when you needed guaranteed and defined behavior, because it was not even possible to unwind the stack.

How to handle exceptions was one of the last topics addressed during the standardization process of C++98. Finding a good solution was not easy, and it took a long time for the following reasons:

1. It was very difficult to determine the degree of safety the C++ standard library should provide. You might argue that it is always best to provide as much safety as possible. For example, you could say that the insertion of a new element at any position in a vector ought to either succeed or have no effect. Ordinarily, an exception might occur while copying later elements into the next position to make room for the new element, from which a full recovery is impossible. To achieve the stated goal, the insert operation would need to be implemented to copy *every* element of the vector into new storage, which would have a serious impact on performance. If good performance is a design goal, as is the case for the STL, you can't provide perfect exception handling in all cases. You have to find a compromise that meets both needs.
2. There is no doubt that it is better to have guaranteed, defined behavior for exceptions without a significant performance penalty instead of the risk that exceptions might crash your system. However, there was a concern that the presence of code to handle exceptions could adversely affect performance. This would contradict the design goal of achieving the best possible performance. During the standardization of C++98, compiler writers stated that, in principle, exception handling can be implemented without any significant performance overhead. However, it turned out that exception specifications could cause performance penalties, so they were replaced by `noexcept` with C++11 (see Section 3.1.7, page 24).

As a result of these discussions, the C++ standard library since C++98 gives the following *basic guarantee* for exception safety:[17] The C++ standard library will not leak resources or violate container invariants in the face of exceptions.

[17] Many thanks to Dave Abrahams and Greg Colvin for their work on exception safety in the C++ standard library and for the feedback they gave me about this topic.

Unfortunately, this is not enough for many purposes. Often, you need a stronger guarantee that specifies that an operation has no effect if an exception is thrown. Such operations can be considered to be *atomic* with respect to exceptions. Or, to use terms from database programming, you could say that these operations support *commit-or-rollback* behavior or are *transaction safe*.

Regarding this stronger guarantee, the C++ standard library now guarantees the following:

- In general, no `erase()`, `clear()`, `pop_back()`, `pop_front()`, or `swap()` function throws an exception. Also, no copy constructor or assignment operator of a returned iterator throws an exception.
- For all *node-based containers* (lists, forward lists, sets, multisets, maps, and multimaps), including the *unordered containers*, any failure to construct a node simply leaves the container as it was. Furthermore, removing a node can't fail, provided that destructors don't throw. However, for multiple-element insert operations of associative containers, the need to keep elements sorted makes full recovery from throws impractical. Thus, all single-element insert operations of associative and unordered containers support commit-or-rollback behavior, provided that the hash function for unordered containers does not throw. That is, the single-element insert operations either succeed or have no effect. In addition, it is guaranteed that all erase operations for both single and multiple elements always succeed, provided that the container's compare or hash function does not throw.

  For lists, even multiple-element insert operations are transaction safe. In fact, all list operations except `remove()`, `remove_if()`, `merge()`, `sort()`, and `unique()` either succeed or have no effect. For some of the exceptional operations, the C++ standard library provides conditional guarantees. Thus, if you need a transaction-safe container, you should use a list.

  For forward lists, `insert_after()`, `emplace_after()`, and `push_front()` are transaction safe.[18]

[18] The C++11 standard does not say this for `emplace_after()` and `push_front()`, which likely is a defect.

- All *array-based containers* with dynamic size (vectors and deques) do not fully recover when an element gets inserted. To do this,

they would have to copy all subsequent elements before any insert operation, and handling full recovery for all copy operations would take quite a lot of time.

Nevertheless, push and pop operations that operate at the (open) end give the strong guarantee. If they throw, it is guaranteed that they have no effect. Note however, that to ensure this behavior, copy constructors instead of move constructors are used for reallocation if move constructors do not guarantee not to throw. Thus, providing a nothrow/noexcept specification for the move operations of the elements will lead to better performance.

Furthermore, if elements have a type with copy and move operations (constructors and assignment operators) that do not throw, every container operation for these elements either succeeds or has no effect.

Note that all these guarantees are based on the requirement that destructors never throw, which should always be the case in C++. The C++ standard library makes this promise, and so must the application programmer.

If you need a container with full commit-or-rollback ability, you should use either a list (without calling or special handling for `remove()`, `remove_if()`, `merge()`, `sort()`, and `unique()`) or an associative/unordered container (without calling their multiple-element insert operations). This avoids having to make copies before a modifying operation to ensure that no data gets lost. Note that making copies of a container could be very expensive.

If you can't use a node-based container and need the full commit-or-rollback ability, you have to provide wrappers for each critical operation. For example, the following function would almost safely insert a value in any container at a certain position:

```
template <typename T, typename Cont, typename Iter>
void insert (Cont& coll, const Iter& pos, const T& value)
{
    Cont tmp(coll);                 // copy container and all elements
    try {
        coll.insert(pos,value);     // try to modify the copy
    }
    catch (...) {                   // in case of an exception
        coll.swap(tmp);             // - restore original container
        throw;                      // - and rethrow the exception
    }
}
```

Note that I wrote "almost," because this function still is not perfect: the `swap()` operation throws when, for associative containers, copying the comparison criterion throws. You see, handling exceptions perfectly is not easy.