Making operations on standard-library containers strongly exception safe

Jyrki Katajainen (University of Copenhagen)

These slides will be made available at `http://www.cphstl.dk`

Environment: STL/C++ standard library

"STL is not a set of specific software components but a set of requirements which components must satisfy."

[Musser & Nishanov 2001]

Element containers:

`vector`
`deque`
`list`
`[multi]set`
`[multi]map`
`hash_[multi]set`
`hash_[multi]map`
`priority_queue`

Generic algorithms:

`copy`
`find`
`nth_element`
`search`
`sort`
`stable_partition`
`unique`
...

Exception safety

An operation on an object is said to be **exception safe** if that operation leaves the object in a valid state when the operation is terminated by throwing an exception. In addition, the operation should ensure that every resource that it acquired is (eventually) released.

A **valid state** means a state that allows the object to be accessed and destroyed without causing undefined behaviour or an exception to be thrown from a destructor.

[Stroustrup 2000, App. E]

Guarantee classification

No guarantee: If an exception is thrown, any container being manipulated is possibly corrupted.

Strong guarantee: If an exception is thrown, any container being manipulated remains in the state in which it was before the operation started. Think of **roll-back semantics** for database transactions!

Basic guarantee: The basic invariants of the containers being manipulated are maintained, and no resources are leaked.

No-throw guarantee: In addition to the basic guarantee, the operation is guaranteed not to throw an exception.

[Stroustrup 2000, App. E]

Our aims

- Provide the strong guarantee of exception safety without any conditions, instead of the basic guarantee or the strong/no-throw guarantees under some specific conditions.

- Focus on dynamic arrays (C++ standard-library `vector`) and associative arrays (C++ standard-library `map` and `multimap`), even if the techniques apply to all standard-library containers!

Background

- It has turned out to be difficult to program strongly exception-safe library components. During the last two years in our "Generic programming" course only one group of students, of about 30 groups in all, has succeeded to provide a strongly exception-safe library component (priority queue or associative array).
- Many of the implementation variations were considered during the C++ standardization process, but not much of this work is documented in the literature.

Current status

- Work in progress!
- Only preliminary experiments carried out so far!
- More implementation work needed!

http://www.cphstl.dk

Big picture

- Basically, there is no efficiency penalty on writing exception-safe code, just more (a lot more) careful programming is required.
- In the worst case the size of the program can grow quadratically.

Unsafe program:	**Strongly exception-safe program:**
statement$_1$	statement$_1$
statement$_2$	if failed, undo statement$_1$ and stop
...	statement$_2$
statement$_k$	if failed, undo statement$_2$, statement$_1$, and stop
	...
	statement$_k$
	if failed, undo statement$_k$, ..., statement$_1$, and stop

What can throw?

In general, all user-supplied functions and template arguments.

```
template <typename E, typename C, typename A>
set<E, C, A>::set(set const&);
```

In this particular case, the following operations can throw an exception:

- function `allocate()` of the allocator (of type `A`) indicating that no memory is available,
- copy constructor of the allocator,
- copy constructor of the element (of type `E`) used by function `construct()` of the allocator,
- invocation of the comparator (of type `C`), and
- copy constructor of the comparator.

What cannot throw?

- Built-in types—including pointers—do not throw exceptions.
- Types without user-defined operations do not throw exceptions.
- Classes with operations that do not throw exceptions.
- Functions from the C library do not throw exceptions unless they take a function argument that does.
- No copy constructor or assignment operator of an iterator defined for a standard container does not throw an exception.

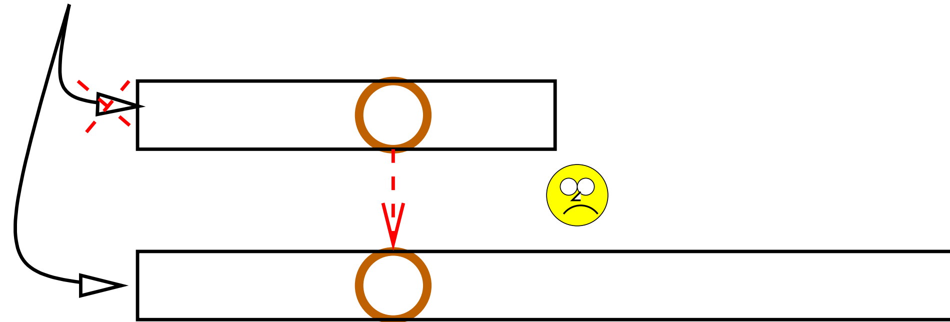
Basically, all classes with destructors that do not throw and which can be easily verified to leave their operands in valid states are friendly for library writers.

Library user's responsibility

The standard library gives **no** guarantees if

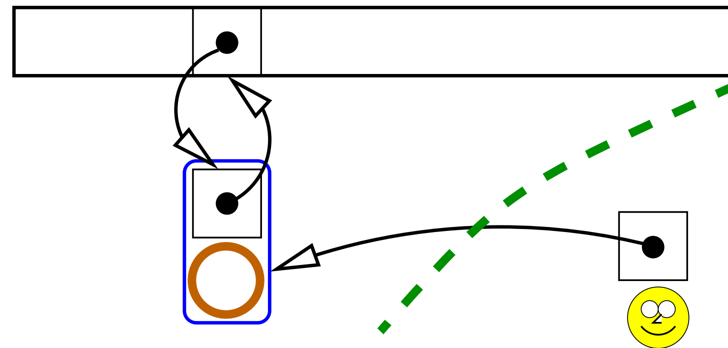
- user-defined operations leave container elements in invalid states,
- user-defined operations leak resources,
- user-supplied destructors throw exceptions, or
- user-supplied iterator operations throw exceptions.

Basic problem with dynamic arrays



- Element copying can fail, but this operation is not necessarily reversible.

Fully exception-safe dynamic arrays



- Instead of storing elements in an array, store pointers to elements.
- Also store a reverse pointer from the elements back to the array to provide iterator support.

Basic problem with associative arrays

- Even if single-element `insert` can be easily made strongly exception safe, it is more difficult to handle multiple-element `insert`.
- For many search trees, like red-black trees, a naive implementation based on repeated insertions does not work since `insert` and `erase` are not each other's mirror images. (That is, rollback might fail!)

Fully exception-safe associative arrays

- Maintain logs of the structural changes made to facilitate a complete rollback. A separate log is maintained for each `insert`.
- Perform all node allocations and element constructions before any updates in the data structure. An exception-safe dynamic array is used for keeping track of the nodes constructed.
- Compact each log by storing the information about structural changes in a bit array. This compaction will reduce the size taken by the logs to $O(m)$ words, where m is the number of elements inserted.

Future work

- Do more experiments [Jens, Mikkel]
  Hypothesis: All standard-library containers can be made exception safe such that all operations on them provide the strong guarantee of exceptions safety.
- Make it possible to combine exception-safe operations easily; implement (software) transactional memory in C++ [Kasper].
- Test whether D supports exception safe programming better than C++.
- Develop a tool to test whether one's code is exception safe or not.

You are welcome to donate your (exception-safe) code to the CPH STL.