

Document number: P0044  
Date: 2015-09-17  
To: SC22/WG21 EWG  
Reply to: David Krauss  
(david\_work at me dot com)

## unwinding\_state: safe exception relativity

`std::uncaught_exception` is now deprecated in favor of `std::uncaught_exceptions`, which appears to be a drop-in replacement. Unfortunately, using it as such defeats its purpose. A class `std::unwinding_state` is proposed to encapsulate the correct usage. Since there is no other use for the current count of unwinding exceptions, it is proposed that the class replace `uncaught_exceptions`, to restore latitude and freedom of extension to implementations.

### 1. Background

Sometimes it is expedient to have the destructor of an automatic object behave differently depending on whether its enclosing scope completed normally or exited by an exception. C++98 included a facility to detect this condition, `std::uncaught_exception`. Its problem is that, during unwinding, scopes can still start and complete normally, and exceptions can still be caught and thrown. It unintentionally turns all unwinding into a different operating environment. The solution is to somehow associate the object's scope with the environment of the destructor.

Fortunately, the Itanium common ABI <sup>1</sup> specifies a global variable counting currently propagating exceptions, and this count can tell an object's destructor whether an exception was thrown (and not yet caught) since its constructor executed. The constructor saves the count, and the destructor again observes the count and compares the two values. If they are equal, then the object's scope was not interrupted.

The problem was identified perhaps even before the standard was ratified. It has been widely known since late 1998, when Sutter published GotW 47, *Uncaught Exceptions*. The solution came only fairly recently. It has appeared in libraries such as Facebook Folly, and others that emulate the `scope(success)` blocks of the D programming language. Such libraries interface directly with the platform runtime library to obtain the `uncaught_exceptions` value.

### 2. Problems

While it is true that `std::uncaught_exceptions` reflects the current practice, there are few advantages and many disadvantages to standardizing it directly.

- Incorrect usage is easier than correct usage: search-and-replacing the new function for the old one appears to work but is a pointless exercise.

---

<sup>1</sup> §EH:2.2.2. Note that its mention of `uncaught_exceptions` is a typo; the text dates to before the function of that name was proposed.

- It may not be the most efficient solution. The question of scope failure or success can be answered without storing an integer, when the sentry object is stored on the stack.
- Correct usage requires the user to declare a nonstatic member variable of integer type. This violates the principle of separation of concerns.
- Incorrect usage cannot be diagnosed, which might otherwise be diagnosable. The strategy only works on strictly scoped objects, and provides incorrect results if an object with dynamic or static lifetime is constructed during unwinding.
- Potentially useful information is thrown away. Identifying a particular exception object could be useful to a debugger or even to the user, but an accounting total cannot do this.

All of these problems are symptomatic of failure to encapsulate. The `uncaught_exceptions` interface does too little, yet it is over-specified.

Some of the above points deserve a closer look.

## 2.1. Temptation

In the current working paper N4527, `uncaught_exceptions` appears exactly twice outside the specifications of itself and its predecessor. In `[ostream::sentry]` §27.7.3.4/4:

```
“ If (os.flags() & ios_base::unitbuf) && !uncaught_exceptions() &&
    os.good() is true...
```

In `[except.throw]` §15.1/7:

```
struct C {
    C() { }
    C(const C&) {
        if (std::uncaught_exceptions()) {
            throw 0; // throw during copy to handler's exception-declaration object (15.3)
        }
    }
};
```

Currently there are two examples of incorrect usage, none of correct usage, and no description of correct usage in `[uncaught.exceptions]` §18.8.4. (As it happens, §18.8.4/2 also mentions comparing the result to zero.) So far, the boilerplate for an `int original_exception_count` member is enough that it's easier to leave them out of the standard. Will users not do the same?

## 2.2. Efficiency

Opportunistic optimizations are possible. Suppose the call stack grows downward, so more deeply nested functions have stack frames at lower addresses. Unwinding tends to move the stack pointer upward, but called destructors move it back downward. An automatic object, allocated on the stack, was created during the current unwinding if and only if its address is lower than the base stack pointer of the last destructor called directly by unwinding. Thus, the address of an empty object can be used to determine its dynamic place in unwinding.

Generally, classes are agnostic to object allocation — `make_unique` is a valid means to a scoped object. However, most sentry objects are invariably allocated on the stack, and it could be worthwhile, for example on an embedded system, to leverage this as a real optimization. A sentry on the stack could be granted smaller layout according to an attribute or only plain alias analysis.

### 2.3. Alternative implementations

An implementation could just as well maintain a linked list of propagating exceptions, with no particular need to count them. The [ARM EHABI](#) already defines an interface for this, rendering the internal `uncaughtExceptions` counter redundant. A lightweight exception handling mechanism for embedded systems (which has been discussed, but not yet designed) may not want the extra global.

A debugger or sanitizer could observe the currently-propagating exception pointer and alert the programmer to particular `throw` statements resulting in allocations or I/O operations during unwinding. This could be taken even further, for example, to track persistent resources relating to aborted operations in a server application.

Estimating all possible interactions with future extensions and ABI implementation strategies is impossible. Good library interface design avoids this by minimizing specificity, answering the necessary question as narrowly and directly as possible.

## 3. Proposal

Remove `std::uncaught_exceptions` and add a class `std::unwinding_state`. The infrastructure of the former is sufficient, but not necessary, for the latter.

```
class unwinding_state {
public:
    unwinding_state() noexcept;
    unwinding_state( unwinding_state const & ) = default;
    unwinding_state & operator= ( unwinding_state const & ) = default;
    ~unwinding_state() = default;

    explicit operator bool() const & noexcept;
};
```

Define the *current unwinding state* as follows:

- For each thread, there is a set of valid unwinding state values. When a thread starts, there is just one such value.
- When an exception is thrown, the current unwinding state takes a new value, distinct from preexisting valid values for the thread.
- When an exception is caught, the former value of the current unwinding state is invalidated. The current state reverts to the value it had before the exception was thrown.

Using an invalid value (or a value which is valid on another thread) results in undefined behavior.

The class is trivially copyable.

```
unwinding_state() noexcept
```

*Effects:* Initializes the current object with the value of the current unwinding state.

```
unwinding_state( unwinding_state const & in ) noexcept
```

*Effects:* Initializes the current object with the value of `in`.

```
unwinding_state & operator = ( unwinding_state const & in ) noexcept
```

*Effects:* Replaces the value the current object with the value of `in`.

```
explicit operator bool() const & noexcept
```

*Returns:* `true` if the value of the object is the value of the current unwinding state.

## 4. Usage, rationale, and implementation

`unwinding_state` is typically used as a nonstatic member in a transaction or guard object. It may be queried in a destructor by using it in a Boolean context.

The accessor function is lvalue-qualified to disallow usage as `if(unwinding_state())`. The class behaves with unsurprising value semantics so that transaction objects may be copyable or movable. A transaction may change its associated scope and expected lifetime by assigning to its `unwinding_state` member.

If a transaction outlives its scope, such as if its processing continues on a different thread, it should be adopted as in `trans->m_uw_state = unwinding_state()` before any operations that could lead to its destruction. Implementations are encouraged to minimize false positives and maximize diagnosis of stale values. Perhaps the facility really belongs inside namespace `this_thread`, but such usage is fairly obscure.

In general, users should be reminded that manually flushing or committing an object at the end of its lifetime may be simpler and safer than doing so automatically in the destructor. Before adding intelligence to a destructor, always weigh the alternatives.

## 5. Conclusion and kudos

We need a specific facility for telling whether two points in program execution are separated by an uncaught `throw`. Given this, `uncaught_exceptions` becomes a liability with no remaining use-cases.

Thanks to Ville Voutilainen for painstakingly clarifying that such mechanisms have applications besides `scope(success)` emulation and throwing destructors, and continuing helpful reviews.

Thanks to Andrey Semashev for review and helpful feedback.