Document Number: N4152

Date: 2014-09-30

Authors: Herb Sutter (hsutter@microsoft.com)

uncaught_exceptions

This paper is a revision of N3614 to implement EWG direction in Bristol.

Motivation

`std::uncaught_exception` is known to be “nearly useful” in many situations, such as when implementing an Alexandrescu-style `ScopeGuard`. [1]

In particular, when called in a destructor, what C++ programmers often expect and what is basically true is: “*uncaught_exception returns true iff this destructor is being called during stack unwinding.*”

However, as documented at least since 1998 in Guru of the Week #47 [2], it means **code that is transitively called from a destructor that could itself be invoked during stack unwinding** cannot correctly detect whether it itself is actually being called as part of unwinding. Once you’re in unwinding of any exception, to `uncaught_exception` everything looks like unwinding, even if there is more than one active exception.

Example 1: Transaction (GotW #47)

Consider this code taken from [2], which shows an early special case of `ScopeGuard` (`ScopeGuard` is described further in the following section):

```
Transaction::~Transaction() {
  if( uncaught_exception() ) // unreliable, ONLY if Transaction could be
    Rollback();              // used from within a dtor (transitively!)
}

void LogStuff() {
  Transaction t( /*...*/ );
    // :::
    // do work
    // :::
} // oops, if U::~U() is called as part of unwinding another exception
  // so uncaught_exception will return true and t will not commit

U::~U() {
  /* deep call tree that eventually calls LogStuff() */
}

// for example:
int main() {
  try {
```

```
    U u;
    throw 1;
  } // U::~U() invoked here
  catch(...) {
  }
}
```

The problem is that, inside `~Transaction`, there is no way to tell whether `~Transaction` is being called as part of stack unwinding. Asking `uncaught_exception()` will only say whether some unwinding is in progress, which might already have been true, rather than answering whether `~Transaction` itself is being called to perform unwinding.

Example 2: ScopeGuard

Alexandrescu's `ScopeGuard` [1, 3] is a major motivating example, where the point is to execute code upon a scope's:

a) termination in all cases == cleanup à la `finally`;
b) successful termination == celebration; or
c) failure termination == rollback-style compensating "undo" code.

However, currently there is no way to *automatically* distinguish between (b) and (c) in standard C++ without requiring the user to explicitly signal successful scope completion by calling a `Dismiss` function on the guard object, which makes the technique useful but somewhere between tedious and fragile. Annoyingly, that `Dismiss` call is also usually right near where the failure recovery code would have been written without `ScopeGuard`, thus not relieving the programmer of having to think about the placement of success/failure determination and compensating actions shouldn't/should occur.

For example, adapted from [1]:

```
void User::AddFriend(User& newFriend)
{
    friends_.push_back(&newFriend);
    ScopeGuard guard([&]{ friends_.pop_back(); });
    :::
    pDB_->AddFriend(GetName(), newFriend.GetName());
    :::
    guard.Dismiss();
}
```

Nevertheless, despite that current drawback, as demonstrated for example in [4], `ScopeGuard` is known to be useful in practice in C++ programs. See slides 35-44 in the Appendix for additional examples from production code.

`ScopeGuard` is desirable because it simplifies control flow by allowing "ad-hoc destructors" without having to write a custom type for each recovery action. For example, in the D programming language, which has language support for `ScopeGuard` in the form of the `scope` statement, the D standard library uses `scope(…)` almost as frequently as `while`.

We would like to enable ScopeGuard and similar uses to automatically and reliably distinguish between success and failure in standard C++ without requiring the user to explicitly signal success or failure by calling a Dismiss function on the guard object. This makes the technique even more useful and less tedious to write code that is clear and exception-safe. The adapted example from [1] would be:

```
void User::AddFriend(User& newFriend)
{
    friends_.push_back(&newFriend);
    ScopeGuard guard([&]{ friends_.pop_back(); });
    :::
    pDB_->AddFriend(GetName(), newFriend.GetName());
    :::
    // no need to call guard.Dismiss();
}
```

Proposal

This paper proposes a simple function that enables the above uses. This paper does not propose adding language support for D-style scope statements, or more general approaches such as suggested in [5].

Option 1: bool unwinding_exception()

The previous version of this paper suggested a function that returned true iff called during stack unwinding. EWG pointed out that this could involve overheads on programs that did not use the feature, because the implementation would have to be ready to answer the query at any time; it might also be an ABI-breaking change in compilers.

Instead, EWG pointed out that having an integer denoting the number of uncaught exceptions was just as useful to implement cases like Transaction and ScopeGuard. Furthermore, Alexandrescu [6] and others point out that this just uses information already present in the major implementations.

Therefore, we do not propose Option 1, favoring instead Option 2 below.

Option 2: int uncaught_exceptions()

This paper proposes a new function **int std::uncaught_exceptions()** that returns the number of exceptions currently active, meaning thrown or rethrown but not yet handled.

A type that wants to know whether its destructor is being run to unwind this object can query uncaught_exceptions in its constructor and store the result, then query uncaught_exceptions again in its destructor; if the result is different, then this destructor is being invoked as part of stack unwinding due to a new exception that was thrown later than the object's construction.

As demonstrated in slides 28-31 of the Appendix [6], this uses information already present in major implementations, where current implementations of ScopeGuard resort to nonportable code that relies on undocumented compiler features to make ScopeGuard "portable in practice" today. This option proposes adding a single new function to expose the information that already present in compilers, so that these uses can be truly portable.

Proposed Wording

In clause 15.5, insert:

15.5.x The std::uncaught_exceptions() function [except.uncaught-exceptions]

1 The function `int std::uncaught_exceptions()` returns the number of exception objects that have been initialized and thrown or rethrown (15.1) but for which no handler has been activated (15.3, 18.8.4).

Acknowledgments

Thanks to Andrei Alexandrescu for prompting this paper and providing examples.

References

[1] A. Alexandrescu. "Change the Way You Write Exception-Safe Code – Forever" (*Dr. Dobb's*, December 2000).

[2] H. Sutter. "Guru of the Week #47: Uncaught Exceptions" (November 1998).

[3] A. Alexandrescu. "Three Unlikely Successful Features of D" (video) (*Lang.NEXT*, April 2012).

[4] K. Rudolph et al. "Does ScopeGuard use really lead to better code?" (StackOverflow, September 2008).

[5] V. Voutilainen. "Accessing current exception during unwinding" (N2952, September 2009).

[6] A. Alexandrescu, "Declarative Control Flow" (*C++ and Beyond*, Stuttgart, Germany, September 2014).

Appendix: [6]

The following is a copy of the handouts of [6], reproduced with permission. In particular, notice in slides 28-31 how the absence of a portable way to get the number of currently active exceptions causes at least some developers to resort to undocumented features that are already present in the major compilers. This proposal is to provide access to this information that already exists in implementations.

Declarative Control Flow

Prepared for The C++ and Beyond Seminar
Stuttgart, Germany, Sep 29–Oct 1, 2014

Andrei Alexandrescu, Ph.D.
andrei@erdani.com

Agenda

- Motivation
- Implementation
- Use cases

⟨action⟩

⟨cleanup⟩
⟨next⟩
⟨rollback⟩

C

```
if (⟨action⟩) {
    if (!⟨next⟩) {
        ⟨rollback⟩
    }
    ⟨cleanup⟩
}
```

C++

```
class RAII {
    RAII() { ⟨action⟩ }
    ~RAII() { ⟨cleanup⟩ }
};
...
RAII raii;
try {
    ⟨next⟩
} catch (...) {
    ⟨rollback⟩
    throw;
}
```


Java, C#

```
⟨action⟩
try {
    ⟨next⟩
} catch (Exception e) {
    ⟨rollback⟩
    throw e;
} finally {
    ⟨cleanup⟩
}
```

Go

```
result, error := ⟨action⟩
if error != nil {
    defer ⟨cleanup⟩
    if !⟨next⟩
        ⟨rollback⟩
}
```


Composition

C

```
if (⟨action₁⟩) {
    if (⟨action₂⟩) {
        if (!⟨next₂⟩) {
            ⟨rollback₂⟩
            ⟨rollback₁⟩
        }
        ⟨cleanup₂⟩
    } else {
        ⟨rollback₁⟩
    }
    ⟨cleanup₁⟩
}
```


C (Pros Only)

```
if (!⟨action₁⟩) {
    goto done;
}
if (!⟨action₂⟩) {
    goto r1;
}
if (!⟨next₂⟩) {
      goto r2;
}
⟨cleanup₂⟩
goto c1;
r2: ⟨rollback₂⟩
⟨cleanup₂⟩
r1: ⟨rollback₁⟩
c1: ⟨cleanup₁⟩
done: ;
```

C++

```
class RAII1 {
    RAII1() { ⟨action₁⟩ }
    ~RAII1() { ⟨cleanup₁⟩ }
};
class RAII2 {
    RAII2() { ⟨action₂⟩ }
    ~RAII2() { ⟨cleanup₂⟩ }
};
...
```


C++

```
RAII1 raii1;
try {
    RAII2 raii2;
    try {
        ⟨next₂⟩
    } catch (...) {
        ⟨rollback₂⟩
        throw;
    }
} catch (...) {
    ⟨rollback₁⟩
    throw;
}
```

Java, C#

```
⟨action₁⟩
try {
    ⟨action₂⟩
    try {
        ⟨next₂⟩
    } catch (Exception e) {
        ⟨rollback₂⟩
        throw e;
    } finally {
        ⟨cleanup₂⟩
    }
} catch (Exception e) {
    ⟨rollback₁⟩
    throw e;
} finally {
    ⟨cleanup₁⟩
}
```


Go

```
result1, error := ⟨action₁⟩
if error != nil {
    defer ⟨cleanup₁⟩
    result2, error := ⟨action₂⟩
    if error != nil {
        defer ⟨cleanup₂⟩
        if !⟨next₂⟩
            ⟨rollback₂⟩
    } else {
        ⟨rollback₂⟩
    }
}
```

Explicit Control Flow = Fail

Declarative Programming

- Focus on stating needed accomplishments
- As opposed to describing steps
- Control flow typically minimal/absent
- Execution is implicit, not explicit
- Examples: SQL, regex, make, config,. . .

- Let's take a page from their book!

According to Seinfeld

Declarative: airplane ticket

Imperative: what the pilot does

Surprising Insight

- Consider bona fide RAII with destructors:

✓ States needed accomplishment?
✓ Implicit execution?
✓ Control flow minimal?

- RAII *is* declarative programming!

More RAII: `ScopeGuard`

- Also declarative
- Less syntactic baggage than cdtors
- Flow is "automated" through placement
- Macro `SCOPE_EXIT` raises it to pseudo-statement status

Pseudo-Statement (C&B 2012 recap!)

```
namespace detail {
   enum class ScopeGuardOnExit {};
   template <typename Fun>
   ScopeGuard<Fun>
   operator+(ScopeGuardOnExit, Fun&& fn) {
      return ScopeGuard<Fun>(std::forward<Fun>(fn));
   }
}

#define SCOPE_EXIT \
   auto ANONYMOUS_VARIABLE(SCOPE_EXIT_STATE) \
   = ::detail::ScopeGuardOnExit() + [&]()
```

Preprocessor Trick (C&B 2012 recap!)

```
#define CONCATENATE_IMPL(s1, s2) s1##s2
#define CONCATENATE(s1, s2) CONCATENATE_IMPL(s1, s2)

#ifdef __COUNTER__
#define ANONYMOUS_VARIABLE(str) \
   CONCATENATE(str, __COUNTER__)
#else
#define ANONYMOUS_VARIABLE(str) \
   CONCATENATE(str, __LINE__)
#endif
```


Use (C&B 2012 recap!)

```
void fun() {
   char name[] = "/tmp/deleteme.XXXXXX";
   auto fd = mkstemp(name);
   SCOPE_EXIT { fclose(fd); unlink(name); };
   auto buf = malloc(1024 * 1024);
   SCOPE_EXIT { free(buf); };

   ... use fd and buf ...
}
```

(if no “;” after lambda, error message is meh)

Painfully Close to Ideal!

```
⟨action₁⟩
SCOPE_EXIT { ⟨cleanup₁⟩ };
SCOPE_FAIL { ⟨rollback₁⟩ }; // nope
⟨action₂⟩
SCOPE_EXIT { ⟨cleanup₂⟩ };
SCOPE_FAIL { ⟨rollback₂⟩ }; // nope
⟨next₂⟩
```

- Note: slide plagiated from C&B 2012

One more for completeness

```
⟨action⟩
SCOPE_SUCCESS { ⟨celebrate⟩ };
⟨next⟩
```

- Powerful flow-declarative trifecta!
- Do not specify flow
- Instead declare circumstances and goals

Can be implemented *today* on ALL major compilers

May become 100% portable:

`http://isocpp.org/files/papers/N3614.pdf`

Credits

- Evgeny Panasyuk: compiler-specific bits
  `github.com/panaseleus/stack_unwinding`

- Daniel Marinescu: `folly` implementation
  `github.com/facebook/folly`

Underpinnings

```
class UncaughtExceptionCounter {
   int getUncaughtExceptionCount() noexcept;
   int exceptionCount_;
public:
   UncaughtExceptionCounter()
    : exceptionCount_(getUncaughtExceptionCount()) {
   }
   bool isNewUncaughtException() noexcept {
      return getUncaughtExceptionCount()
         > exceptionCount_;
   }
};
```

- Only detail left: `getUncaughtExceptionCount()`

gcc/clang

```
inline int UncaughtExceptionCounter::
getUncaughtExceptionCount() noexcept {
   // __cxa_get_globals returns a __cxa_eh_globals*
   // (defined in unwind-cxx.h).
   // The offset below returns
   // __cxa_eh_globals::uncaughtExceptions.
   return *(reinterpret_cast<int*>(
      static_cast<char*>(
         static_cast<void*>(
            __cxxabiv1::__cxa_get_globals()))
               + sizeof(void*)));
}
```


gcc/clang

```
namespace __cxxabiv1 {
   // defined in unwind-cxx.h from from libstdc++
   struct __cxa_eh_globals;
   // declared in cxxabi.h from libstdc++-v3
   extern "C"
   __cxa_eh_globals* __cxa_get_globals() noexcept;
}
```

MSVC 8.0+

```
struct _tiddata;
extern "C" _tiddata* _getptd();

inline int UncaughtExceptionCounter::
getUncaughtExceptionCount() noexcept {
   // _getptd() returns a _tiddata*
   //      (defined in mtdll.h).
   // The offset below returns
   //    _tiddata::_ProcessingThrow.
   return *(reinterpret_cast<int*>(static_cast<char*>(
         static_cast<void*>(_getptd()))
            + sizeof(void*) * 28 + 0x4 * 8));
}
```


Layering

```
template <typename FunctionType, bool executeOnException>
class ScopeGuardForNewException {
   FunctionType function_;
   UncaughtExceptionCounter ec_;
public:
   explicit ScopeGuardForNewException(const FunctionType& fn)
         : function_(fn) {
   }
   explicit ScopeGuardForNewException(FunctionType&& fn)
         : function_(std::move(fn)) {
   }
   ~ScopeGuardForNewException() noexcept(executeOnException) {
      if (executeOnException == ec_.isNewUncaughtException()) {
         function_();
      }
   }
};
```

Icing

```
enum class ScopeGuardOnFail {};

template <typename FunctionType>
ScopeGuardForNewException<
   typename std::decay<FunctionType>::type, true>
operator+(detail::ScopeGuardOnFail, FunctionType&& fn) {
   return
      ScopeGuardForNewException<
            typename std::decay<FunctionType>::type, true>(
         std::forward<FunctionType>(fn));
}
```


Cake Candles

```
enum class ScopeGuardOnSuccess {};

template <typename FunctionType>
ScopeGuardForNewException<
   typename std::decay<FunctionType>::type, false>
operator+(detail::ScopeGuardOnSuccess, FunctionType&& fn) {
   return
      ScopeGuardForNewException<
            typename std::decay<FunctionType>::type, false>(
         std::forward<FunctionType>(fn));
}
```

Use Cases

Tracing

```
void login() {
   SCOPE_FAIL {
      cerr << "Failed to log in.\n";
   };
   ...
}
```

- User-displayable (unlike stack traces)
- Show only major failure points

Transactional Work

```
void buildFile(const string& name) {
   auto tmp = name + ".deleteme";
   auto f = fopen(tmp.data(), "w");
   enforce(f, "...");
   SCOPE_SUCCESS {
      enforce(fclose(f) == 0, "...");
      rename(tmp.data(). name.data());
   };
   SCOPE_FAIL {
      fclose(f); // don't care if fails
      unlink(tmp.data());
   };
   ...
}
```


Order Still Matters

```
void buildFile(const string& name) {
   auto tmp = name + ".deleteme";
   auto f = fopen(tmp.data(), "w");
   enforce(f, "...");
   SCOPE_FAIL { // PLANTED TOO EARLY!
      fclose(f); // don't care if fails
      unlink(tmp.data());
   };
   SCOPE_SUCCESS {
      enforce(fclose(f) == 0, "...");
      rename(tmp.data(). name.data());
   };
   ...
}
```

- Handler "sees" exceptions after planting

Please Note

Only `SCOPE_SUCCESS` may throw

Postconditions

```
int string2int(const string& s) {
   int r;
   SCOPE_SUCCESS {
      assert(int2string(r) == s);
   };
   ...
   return r;
}
```

Changing of the Guard

```
void process(char *const buf, size_t len) {
   if (!len) return;
   const auto save = buf[len - 1];
   buf[len - 1] = 255;
   SCOPE_EXIT { buf[len - 1] = save; };
   for (auto p = buf;;) switch (auto c = *p++) {
      ...
   }
}
```


Scoped Changes

```
bool g_sweeping;
void sweep() {
   g_sweeping = true;
   SCOPE_EXIT { g_sweeping = false; };
   auto r = getRoot();
   assert(r);
   r->sweepAll();
}
```

No RAII Type? No Problem!

```
void fileTransact(int fd) {
   enforce(flock(fd, LOCK_EX) == 0);
   SCOPE_EXIT {
      enforce(flock(fd, LOCK_UN) == 0);
   };
   ...
}
```

- No need to add a type for occasional RAII idioms

Remarks

- All examples taken from production code
- Declarative focus
  - Declare contingency actions by context
- `SCOPE_*` more frequent than `try` in new code
- The latter remains in use for actual handling
- Flattened flow
- Order still matters

Summary

Summary

- SCOPE_EXIT
- SCOPE_FAILURE
- SCOPE_SUCCESS