Document Number: N3614

Date: 2013-03-11

Authors: Herb Sutter (hsutter@microsoft.com)

unwinding_exception

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: 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() was being called as part of unwinding another exception
  // uncaught_exception will "erroneously" 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 key 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, not whether `~Transaction` 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. Further, it leads to simpler code, as shown by the following example from D…

Example 3: D scope statement

The following side-by-side code example (drawn from [3]) is written in the D programming language, which has language support for `ScopeGuard` in the form of the `scope` statement. Note the simplification achieved by the D `scope(exit)` and `scope(failure)` statements:

Example

```
void[] read(string name)
{
    invariant fd = std.c.linux.linux.open(toStringz(name), O_RDONLY);
    cenforce(fd != -1, name);
    scope(exit) std.c.linux.linux.close(fd);

    struct_stat statbuf = void;
    cenforce(std.c.linux.linux.fstat(fd, &statbuf) == 0, name);

    void[] buf;
    auto size = statbuf.st_size;
    if (size == 0)
    {  /* The size could be 0 if the file is a device or a procFS file,
        * so we just have to try reading it.
        */
        int readsize = 1024;
        while (1)
        {
            buf = GC.realloc(buf.ptr, size + readsize, GC.BlkAttr.NO_SCAN)
                [0 .. cast(int)size + readsize];
            enforce(buf, "Out of memory");
            scope(failure) delete buf;

            auto toread = readsize;
            while (toread)
            {
                auto numread = std.c.linux.linux.read(fd, buf.ptr + size, toread);
                cenforce(numread != -1, name);
                size += numread;
                if (numread == 0)
                {   if (size == 0)                      // it really was 0 size
                        delete buf;                     // don't need the buffer
                    return buf[0 .. size];              // end of file
                }
                toread -= numread;
            }
        }
    }
    else
    {
        buf = GC.malloc(size, GC.BlkAttr.NO_SCAN)[0 .. size];
        enforce(buf, "Out of memory");
        scope(failure) delete buf;

        cenforce(std.c.linux.linux.read(fd, buf.ptr, size) == size, name);

        return buf[0 .. size];
    }
}
```

```
void[] read(string name)
{
    immutable fd = std.c.linux.linux.open(toStringz(name), O_RDONLY);
    cenforce(fd != -1, name);
    scope(exit) std.c.linux.linux.close(fd);

    struct_stat statbuf = void;
    cenforce(std.c.linux.linux.fstat(fd, &statbuf) == 0, name);

    immutable initialAlloc = statbuf.st_size ? statbuf.st_size + 1 : 1024;
    void[] result = GC.malloc(initialAlloc, GC.BlkAttr.NO_SCAN)
        [0 .. initialAlloc];
    scope(failure) delete result;
    size_t size = 0;

    for (;;)
    {
        immutable actual = std.c.linux.linux.read(fd, result.ptr + size,
                result.length - size);
        cenforce(actual != actual.max, name);
        size += actual;
        if (size < result.length) break;
        auto newAlloc = size + 1024 * 4;
        result = GC.realloc(result.ptr, newAlloc, GC.BlkAttr.NO_SCAN)
            [0 .. newAlloc];
    }

    return result[0 .. size];
}
```

© 2012 Andrei Alexandrescu

Proposal

This paper does not propose adding language support for D-style `scope` statements.

Instead, it proposes a new function **`std::unwinding_exception`** that returns `true` iff we are executing a destructor of a stack-based object that is being called to perform stack unwinding.

This enables `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. 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();
```

```
}
```

This code would now work the same way whether transitively invoked from a destructor or not.

Proposed Wording

In clause 15.5, insert:

15.5.x The std::unwinding_exception() function [except.unwinding]

1 The function `bool std::unwinding_exception()` returns `true` if called inside a destructor body and the destructor is being invoked to perform stack unwinding, and returns `false` otherwise.

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).