Document: D????
Date: 2019-10-27
Reply-to: Lisa Lippincott <lisa.e.lippincott@gmail.com>
Audience: EWGI

Throw Goto

Lisa Lippincott

Abstract

When in the course of program execution an operation in progress is found to be unnecessary and wasteful, the operation should be canceled and its stack unwound. Exception objects are ill-suited to this unwinding, for reasons explained within. A minimal mechanism for unwinding the stack to a particular point is here proposed, introducing a new form of exception, similar in nature to a stack unwinding `longjmp`. As this mechanism combines the locality of `throw` with the structure of `goto`, it is named `throw goto`.

1 A Pattern for Cancellation

Cancellations and errors both involve exit from a function before the task of the function is complete. In this paper, any such early exit will be called *exceptional exit* .[1] While the stack unwinding process of exceptional exit is identical for both cancellations and errors, the design patterns governing the initiation and completion of exceptional exit differ.

Exceptional exit with an error is governed by the strategy failure pattern: when a function determines that it cannot complete its low-level goal, it exits exceptionally. The exceptional exit is completed by a handler in a higher-level function which selects an alternate strategy toward its own high-level goal. The nature of the original error may occasionally influence the choice of alternate strategy; that information is communicated through an exception object.

Exceptional exit for cancellation is governed by a design pattern we may call *serendipitous success:* when a higher-level goal is met before a function has achieved its low-level goal, the function may exit exceptionally in order to avoid continuing wastefully. No alternate strategy is needed; the exceptional exit is completed by a handler that notes the success of the high-level goal. No information from the canceled function need be provided to the handler; its work has been abandoned.

The similarities between the strategy failure pattern and the serendipitous success cancellation pattern suggest that both patterns exit through what we may rightly call exceptions, but the differences suggest that a second form of exception be introduced for cancellation. Such an exception would

1. be thrown from a point specified in a low-level function, but only under a success condition specified by a higher-level function, and

2. be caught only by the handler corresponding to the satisfied success condition.

In the general case, the first property seems to require an indirection that brings with it some inefficiency: if the success condition cannot be inlined into the lower-level function, optimization opportunities will be lost. To leave no room for a lower-level language, this proposal does not address the first property, leaving it to a a higher-level library or language mechanism. Instead, we here propose *targeted exceptions* that may be used in a way satisfying the second condition.

[1]Under this definition, destruction of an incompletely-executed coroutine object is also exceptional exit.

2 `throw goto; catch goto`

Targeted exceptions are thrown with a new form of *throw-expression*, and caught with a new form of *handler*. Both are distinguished by the keyword `goto`.

throw-expression:
    `throw goto`$_{opt}$ *assignment-expression*

handler:
    `catch (` *exception-declaration* `)` *compound-statement*
    `catch goto` *identifier* `:` *compound-statement*

The identifier in a `catch goto` handler names an object of type `std::exception_target` and automatic storage duration. The scope of the identifier and the lifetime of the object extend throughout, but not beyond, the *compound-statement* of the *try-block* to which the handler is attached; for a *function-try-block*, the scope and lifetime also extend throughout the *ctor-initializer*.

Correspondingly, the *assignment-expression* in a throw goto expression is converted to `exception_-target&`, and the expression transfers control to the handler associated with the indicated object.

```
void foo()
  {
   try
     {
      bar( success_target );
     }
   catch goto success_target:
     {
      cout << "Success!\n";
     }
  }
```

```
void bar( std::exception_target& target )
  {
   while ( incomplete() )
     {
      do_stuff();
      if ( serendipitous_success_of_foo() )
         throw goto target;
      do_more_stuff();
     }
  }
```

3 Details

3.1 Suspendable exception targets

A *try-block* or *function-try-block* is *suspendable* if its *compound-statement* or *ctor-initializer* contains an *await-expression* or *yield-expression*. An `exception_target` object is *suspendable* if it is associated with a handler of a suspendable *try-block* or *function-try-block*. The object is suspended when, during its lifetime, the enclosing coroutine is suspended. When an `exception_target` object is not suspended, it is *active*.

An active `exception_target` object has an *activating thread*. The activating thread of a nonsuspendable `exception_target` object is the thread on which the object was created. The activating thread of an active suspendable `exception_target` object is the thread on which the enclosing coroutine was most recently activated. A suspended `exception_target` object has no activating thread.

3.2 The `exception_target` type

Objects of type `exception_target` are created and destroyed only by language mechanisms; the type has no publicly-accessible constructors, destructors, or assignment operators. This creates a separation of usage: an `exception_target` cannot be thrown by a `throw` expression without `goto`, and only types convertible to `exception_target&` may be thrown with `throw goto`. Programs may freely use references and pointers to `exception_target` objects.

Nonsuspendable `exception_target` objects created on a thread are naturally ordered by length of jump. It seems churlish to hide this ordering from users. Instead, `exception_target` objects are totally ordered by `<=>` in a manner consistent with these rules:[2]

[2]The intent is that for many implementations, either address order or reverse address order will suffice.

- a and b are equal if and only if a and b are the same object.
- Notwithstanding the rules below, if a and b are `exception_target` objects created within different function invocations, and either a or b is suspendable, the ordering of a and b is stable but unspecified.
- If a and b were created by different entries into *try-blocks* on the same thread, the elder of a or b is greater.
- If a and b were created by the same entry into a *try-block*, but associated with different handlers, the object associated with the textually later handler compares greater.

[*Example:*

```
try
  { /* ... */ }
catch goto lesser_target:
  { /* ... */ }
catch goto greater_target:
  { /* ... */ }
```

```
try
  {
   try
     { /* ... */ }
   catch goto lesser_target:
     { /* ... */ }
  }
catch goto greater_target:
  { /* ... */ }
```

—*end example*]

3.3 Undefined Behavior

The `throw goto` expression has undefined behavior if the expression does not refer to an extant `exception_target` object or if the current thread is not the activating thread of the `exception_target` object.

3.4 Stack Unwinding

The stack unwinding process is identical for the two kinds of exceptions. Functions marked `noexcept` terminate if they exit with either kind of exception. Both kinds of exceptions are counted by the function `uncaught_exceptions` and, ideally, any zombie `uncaught_exception` function.

3.5 `catch (...)`

Exceptions thrown with `throw goto` are handled only by the indicated `catch goto` handler; they are not caught by `catch (...)`. This preserves the existing usage of `catch (...)` to respond to all failures. It does not, however, preserve the usage of `catch (...)` for cleanup. As has been best practice for many years, destructors should used for cleanup.