Cancellation is not an Error

Document #: P1677
Date: 2019-05-18
Project: Programming Language C++
SG1 Concurrency and Parallelism, LEWG Library Evolution, EWG Evolution
Reply-to: Kirk Shoop
<kirkshoop@fb.com>

Contents

1 Introduction

One of the basis operations for any async function is cancellation. In this paper we explore the uses of cancellation to determine how to represent the result of a cancelled async function.

In the `jthread` paper [P0660R9] a mechanism for sending a cancellation signal is described. A `stop_source` allows cancellation to be requested. A callback can be attached to the corresponding `stop_token` that will be called when the cancellation is requested. The `stop_token` also has methods to report the current cancellation state.

The `stop_source`/`stop_token` mechanism provides a way to request an async function to stop but does not specify how the async function completes without a result. This paper will explore how an async function will complete when it is stopped.

> NOTE: This paper does not depend on a particular representation of an async function. async functions may return Futures, Executors, Senders, Awaitables or something completely different. While this paper may use some of these representations in example code, they are used for exposition only.

2 Motivation

Motivations for this paper include previously proposed features (eg. jthread), existing practice (eg. Callbacks), and the needs of generic code and algorithms (eg. Algorithms that cancel).

2.1 jthread

[P0660R4] is an earlier revision of the `jthread` paper that defined a `std::interrupted` exception and a `std::this_thread::throw_if_interrupted()` API. These were intended to exit an arbitrary scope using the exception mechanism.

This revision of the paper was discussed in an SG1 meeting in Seattle wiki. After several issues were described related to TLS and reporting cancellation as an exception, the participants voted that the parts related to the `std::interrupted` exception should be removed from the paper.

The issues related to reporting cancellation as an exception included explicitly ignoring `std::interrupted` and transporting `std::interrupted`.

2.1.1 explicitly ignoring `std::interrupted`

[P0660R4] added this to `std::thread`.

> An uncaught interrupted exception in the started thread of execution will silently be ignored. [Note: Thus, an uncaught exception thrown by this_thread::throw_if_interrupted() will cause the started thread to end silently. — end note]

This is an example of how existing error handling must change when cancellation is reported as an error. `std::interrupted` requires that in every function on the stack in the thread at the time cancellation is reported, the `std::interrupted` is explicitly ignored. `std::interrupted` was intended to be implicitly ignored, and to help achieve this, `std::interrupted` was not derived from `std::exception`. Explicit handling of `std::interrupted` is still required in that:

— noexcept functions cannot be on the stack

— all functions that cross ABI boundaries, such as callbacks passed to C functions, like OS APIs, must suppress `std::interrupted`
— all `catch(...)` must rethrow, just in case the exception is `std::interrupted`
— all `catch(...)` must be called for `std::interrupted`, since many `catch(...)` are used to cleanup and some of those instances are in std lib implementations.
— all `catch(const std::interrupted&)` must rethrow.

It is interesting to note that the ABI boundary restriction conflicts with the catch restrictions. The ABI and the catch restrictions also led to the second issue.

2.1.2 transporting `std::interrupted`

`std::exception_ptr` and `std::current_exception()` were introduced to support async facilities like Futures, Executors and Coroutines that must be able to transport exceptions from one thread to another and facilities that transport exceptions across ABI boundaries. While this appears to satisfy the catch restrictions by re-throwing a saved `std::exception_ptr` on a different thread, this adds even more instances of code that need to be explicitly aware of `std::interrupted`.

When `std::interrupted` is transported from thread A to thread B: - was `std::interrupted` intended to tear down thread A? How is that determined? - does thread B support `std::interrupted` (it might be an OS thread)? - does every function on the stack in thread B when the exception is re-thrown support `std::interrupted`?

2.2 Algorithms that cancel

There are many algorithms for async functions. These algorithms must be able to trigger cancellation and stop cleanly when cancelled.

Some examples:

— the `when_any()` (aka `amb()`) algorithm which cancels the other producers once one of them produced a result (and emits no error).
— the `when_all()` (aka `zip()`) algorithm which cancels the other producers when one completes with an error.
— the `switch_on_value()` algorithm which cancels the previous producer and in favor of the new producer (and only emits an error from the current producer).
— the `take_until()` algorithm which cancels a source when a trigger completes and cancels a trigger when the source completes.
— the `timeout()` algorithm which cancels a source when it does not produce a value before the timeout and then emits `timeout_error` (which is defined as part of the `timeout()` algorithm).

`when_all()` and `when_any()` are expected to be the most familiar and the rest have similar needs.

2.2.1 `when_any()`

One expression of the `when_any()` algorithm takes a set of async functions that have a common result type and returns the result of the first async function to complete with a result or error and cancels the rest and emits the result or error.

```
void foo() {
  common_type_t<invoke_result_t<f>, invoke_result_t<g>> v = wait(when_any(f, g));
  // ..
}
```

`when_any()` must have a way to know when an async function completed. `when_any()` is interested in knowing when an async function has completed with a result, with an error and with neither. When all the async functions complete with neither a result nor an error, then `when_any()` must complete with neither a result nor error.

2.2.2 `when_all()`

One expression of the `when_all()` algorithm takes a set of async functions and returns a tuple with an element that contains the result of each async function. The first async function to complete with an error will cancel the remaining async functions and emit the error.

```
void foo() {
  tuple<invoke_result_t<f>, invoke_result_t<g>> v = wait(when_all(f, g));
  // ..
}
```

`when_all()` must have a way to know when an async function completed. `when_all()` is interested in knowing when an async function has completed with a result, with an error and with neither. When any of the async functions complete with neither a result nor an error, then `when_all()` must complete with neither a result nor an error.

2.3 Callbacks

As the most common pattern for expressing async, callbacks also need to be called with neither-a-result-nor-an-error. There is a lot to be said about callbacks and [P1678] is focused on callbacks. The following will cover only some of that larger topic.

Examples of callbacks can be found in the networking TS [N4771]. The completion signature for `async_accept()` is `void(error_code ec, socket_type s)`. This signature clearly displays that the first argument is used for the error channel and that the second argument is used for the value channel. Perhaps, if the completion is an object, the destructor of that object might be a signal that there was neither a result nor an error.

2.3.1 destructor style

There are reasons not to use the destructor to signal that there was neither a result nor an error.

The primary reason is that the compiler calls the destructor for end-of-lifetime which includes exception unwind and success unwind and unwind of a moved-from object. If the destructor is considered a signal to the Callback, then the meaning for exception unwind is *ignore* and success unwind is 'neither a result nor an error' and moved-from object unwind is *ignore*. This would force Callback destructors to handle the two cases explicitly by maintaining state; "was error() called?", "was value() called?", "is the object moved-from?". Also, using the destructor to signal 'neither a result nor an error' leaves blocking as the only option for holding the lifetime of the current object for the end of some other nested or dependent async function. The state and blocking implications are both great reasons to avoid using the destructor for the 'neither a result nor an error' signal. But there is another, async vs object lifetime.

2.3.2 value and error arguments style

Using separate arguments to a callback to represent error and value channels involves some unfortunate tradeoffs. The completion signature `void(error_code ec, socket_type s)` for `async_accept()` in [N4771] implies that the `socket_type` must support an invalid or empty state when ec contains an error. This style requires that all the parameters used in a completion signature support invalid or empty states, because the same function will be called for error and success. This requires all implementations of callbacks to check the arguments for validity before using the arguments. These checks introduce branches, which can be particularly expensive instructions.

Another way to represent this is to use `std::optional` explicitly on all the args so that the value types used as callback arguments are not required to support an invalid or empty state.

NOTE: The error_code supports an empty state. The empty state for an error_code is the success code.

2.3.3 `std::expected` style

Another callback pattern is to combine the value and error into one argument. The completion signature for the `async_accept()` example might change to look something like `void(expected<error_code, socket_type> e)`.

This style does not require `socket_type` to support an invalid or empty state because it does not need to be constructed when there is an error. The branches required by the value and error arguments style are still required in this style, because the same function will be called for error and success.

There is also an additional cost in the codegen for packing and unpacking `std::expected`. The cost for `std::expected` is not as bad as when the value is a `std::tuple` or a `std::variant` of `std::tuple`s, but still worse than when it is an plain argument to the function. For instance, something that transforms the result from one type to another has to check the error, unpack the result or error and repack the transformed result or original error into the outgoing expected type.

2.3.4 multiple function style

Some of the tradeoffs encountered when mixing errors and results into the same 'channel' (where function arguments and function results are both channels for communication with a function), motivated the creation of the C++ exception channel. C++ exceptions do not require the implementation of a function to check for the validity of function return values before using them and do not require that function return values support invalid or empty states (basically re-implementing `std::optional` in each type) nor require the use of types that combine error/value alternatives like `std::expected`.

Using multiple functions for error and result is equivalent to the separation of `return value` and `throw`/`catch` in the language. Using multiple functions for error and result produces very different tradeoffs than when mixing error and result together in one function. The [`std::promise` type:] is an example of using multiple functions for error and result that already exists.

A challenge with the [`std::promise` type:] is that it is a type with only one implementation, whereas callbacks are intended to be a concept or signature with many implementations. There are several examples of concepts that use multiple functions for error and result. These concepts primarily differ only in the names of the concepts and the names of the functions.

— Reactive Extensions defines the Observer concept which has been implemented in many different languages including C++. The rxcpp implementation uses the names `Observer::on_next(T)`, `Observer::on_error(std::exception_ptr)` and `Observer::on_completed()`

— [P1055R0] defines the Single concept using the names `Single::value(T)`, `Single::error(E)` and `Single::done()`
— [P1341R0] defines the Receiver concept using the names `Receiver::value(Tn...)`, `Receiver::error(E)` and `Receiver::done()`. The pushmi library has an implementation of the Receiver concept.
— [P1660] defines the Callback concept that subsumes the Invocable and Fallback concepts resulting in the names `Invocable::operator()(Tn...)`, `Fallback::error(E)` and `Fallback::done()`. [P1660] includes an example implementation.

The Callback concept defined in [P1660] has been gaining support in SG1 recently. A completion object for the `async_accept()` example might change to look something like:

```
struct async_accept_completion {
  void operator()(socket_type s) && noexcept;
  void error(error_code) && noexcept;
  void error(exception_ptr) && noexcept;
  void done() && noexcept;
};
```

Where:

— `operator()` is only called for success
— `error()` is only called for failure
— `done()` is only called for neither-a-result-nor-an-error

Provides:

— each function can be specified to be called on a different execution agent
— value types do not need to represent invalid or empty states
— none of the functions are required to add branches and checks for errors or validity
— all types are passed as function arguments with no required packing/unpacking
— overloads of each method allow different types to be supported without use of `std::variant`
— overloads of each method allow different numbers of arguments to be supported without use of `std::optional` or `std::variant<std::tuple<>...>`

2.3.5 gratuitous

Note: This is for those that object to named methods on an Invocable object.

In an imaginary world these could be renamed as operators in the language. Say that:

— `void error(E)` became `void operator catch(E)`
— `void done()` became `void operator break return()`
— where `catch(callback, std::current_exception());` called `callback.operator catch(std::current_exception`
— where `break return (callback);` called `callback.operator break return();`

Staying with the `async_accept` example `async_accept_completion` might look like this:

```
struct async_accept_completion {
  void operator()(socket_type s) && noexcept;
  void operator catch(error_code) && noexcept;
  void operator catch(exception_ptr) && noexcept;
  void operator break return() && noexcept;
};
```

other capabilities of `break return` are imagined in scope_success, scope_fail, scope_done blocks

2.4 Exception noise

Cancellation is very common when using async functions. Reporting cancellations as exceptions creates a lot of noise because cancellation is expected to occur frequently.

This noise affects logging and debugging and other forms of analysis. exceptions used to report cancellation have to be filtered or categorized in many different tools and libraries to control for that noise.

2.5 sync functions (not a typo)

sync functions also need to complete with neither a result nor an error.

The clearest expression of this involves coroutines and generators. Another example is `std::optional`.

2.5.1 coroutine generator

This example is also made clearer by avoiding Iterators.

```cpp
template<class T>
struct generator {
  task<T> next() {..}
};

generator<int> fortyTwos() {
  for (int i = 0; i < 5; ++i) {
    // the co_await g.next()
    // resumes with the int 42
    co_yield 42;
  }

  // the co_await g.next()
  // resumes with?
  co_return;
}

// assumes that co_await g.next()
// completes with neither a result
// nor an error
task<void> foo() {
  auto g = fortyTwos();
  for(;;) {
    auto fortyTwo = co_await g.next();
  }
}
```

One thing that is not immediately obvious in the example is that types like `generator<int>` actually create a coroutine whose body is allowed to produce two different result types. `co_yield 42;` resolves the matching `co_await g.next()` with an int while `co_return;` resumes the matching `co_await g.next()` with void.

Obviously `next()` returning `task<int>` and `task<void>` does not work in C++ today, which is why generator must model something more complicated like a Range where `begin()` and `operator++()` both produce new iterators that are either a proxy to the yielded result or compare equal to `end()` when 'void' is returned.

While this seems natural for Range, it also affects `std::optional`.

2.5.2 `std::optional`

Range (with size 0|1), `std::optional` and even `std::variant<std::monostate,..>` are ways to model optional values in C++. They are themselves values that provide access to a value or nothing.

It might seem that if cancellation is not an error that `std::optional` would allow cancellation to be composed into the result rather than as an exception. This path was rejected previously because of the impact that it would have on code. all results for all functions that could be cancelled or would use functions that could be cancelled would have to return `std::optional`. All callers of functions that returned `std::optional` would have to explicitly check, extract the result or forward on the empty result. This wrapping and unwrapping is expensive at runtime and messy in the code and very error prone (the cancellation may not propagate when it should). These are all reasons that C++ exceptions have a separate channel and thus motivate a separate channel for neither-a-result-nor-an-error.

An imaginary world, where a sync function can complete with neither-a-result-nor-an-error, would have cleaner code.

Real

```cpp
std::optional<int> op() {
  if (!has_feature()) {
    return {};
  }
  return feature();
}

void foo() {
  // ..
  for(int done = false;!done;done=true){
    auto i = op();
    if (!i) {
      break;
    }
    // use *i..
  } // jumps here when the feature is
    // not supported
  // ..
}
```

Imaginary

```cpp
int op() {
  if (!has_feature()) {
    break return;
  }
  return feature();
}

void foo() {
  // ..
  {
    auto i = op();
    // use i..
  } // jumps here when the feature is
    // not supported
  // ..
}
```

3 Conclusions

The cancellations, covered in Motivation above, are not errors and the functions that were cancelled should complete with neither-a-result-nor-an-error.

Further, cancellation is not the only case covered in Motivation above, where a function would benefit from completing with neither-a-result-nor-an-error.

Finally, neither-a-result-nor-an-error is a signal that does not have a good representation using the existing forms of function output.

3.1 Function output

Here is a short description of the options currently in the language for functions to return values. These options boil down to three channels; return value, out-parameter arguments, and throwing exceptions.

3.1.1 Values

In C, there are three ways to communicate a result:

— return a value
— set value(s) into out-parameter(s)
— call a parameter, that is a function, with arguments(s)

3.1.2 Exceptions

C++ added a third mechanism for communicating a result - throwing exceptions. Adding exception throwing as a separate communication channel allowed code to focus on the path of success and delegate the responsibility for exception handling to the caller by default. C++ made support for exceptions implicit. Functions do not have a mechanism to opt-in to exception support. Functions can opt out of emitting exceptions using `noexcept`, but the compiler still is responsible for ensuring that an attempt to throw an exception in a `noexcept` function will result in a call to `std::terminate`.

3.1.3 Multiplexing

These mechanisms can be multiplexed and de-multiplexed, with additional overhead in code size and runtime.

Examples of mux for return values and out-parameters:

— `optional<T>` allows return without a result.
— `expected<E, T>` allows an error to be returned without an exception.
— `expected<E, optional<T>>` allows an error to be returned without an exception and for nothing to be returned.
— `expected<optional<variant<tuple<Tn0...>, tuple<Tn1...>, ..>>, E>` allows the parameters that are supported by one of an overload set of callback functions to be returned as a value and an error to be returned without an exception and for nothing to be returned.

Potential syntax to simplify the code that needs to be written to demux these values can be found in the proposal for pattern matching [P1371R0].

NOTE: while `expected`, `variant` and `tuple` all correspond to C++ language features (exception & return value `expected`, overload set of functions `variant`, and multiple arguments to a function `tuple`), `optional` does not have a language representation. Pointer is not a language representation as`optional` is a super-set of Pointer, because `optional` stores the value when it is valid, while Pointer does not.

4 Proposals

There are designs that can support result and error and neither-a-result-nor-an-error for both library and language.

4.1 Library

When adding async functions to the library there must be a way to represent a result and an error and neither-a-result-nor-an-error.

Currently the ways to represent result and error were covered in `std::optional`, coroutine generator, Callbacks and jthread above. Of these, the only one with a working solution for a result and an error and neither-a-result-nor-an-error is `async_accept_completion` in Callbacks. Reproduced here for convenience:

```
struct async_accept_completion {
  void operator()(socket_type s) && noexcept;
  void error(error_code) && noexcept;
  void error(exception_ptr) && noexcept;
  void done() && noexcept;
};
```

Where:

— `operator()` is only called for success
— `error()` is only called for failure
— `done()` is only called for neither-a-result-nor-an-error

Provides:

— each function can be specified to be called on a different execution agent
— value types do not need to represent invalid or empty states
— none of the functions are required to add branches and checks for errors or validity
— all types are passed as function arguments with no required packing/unpacking
— overloads of each method allow different types to be supported without use of `std::variant`
— overloads of each method allow different numbers of arguments to be supported without use of `std::optional` or `std::variant<std::tuple<>>`

4.2 Language

As hinted in `std::optional` above, there is no language feature that supports neither-a-result-nor-an-error. Here are some thoughts on what this might look like in the language.

4.2.1 co_done & catch_co_done

One option is to tie this to coroutines, and add `co_done` to emit the signal, `operator co_done()` to customize the signal and `try {} catch_co_done() {}` to intercept the signal.

Pros: familiar to coroutines

Cons:

```
- limits usage to coroutines
- explicit scope
- requires adding try blocks to intercept a signal that is not an error
```

4.2.2 scope library

Another option is to provide a new model for handling implicit signals in a scope.

There is a library that is adding a new model for handling implicit signals in a scope. The scope library [P0052R10] introduces `scope_exit`, `scope_fail` and `scope_success`. These are used to introduce new implicit scopes (no braces required) and invoke a function at the end of that scope.

The paper contains a simple example:

```
void grow(vector<int>& v){
  scope_success guard([]{ cout << "Good!" << endl; });
  v.resize(1024);
}
```

Pros:

- — familiar library
- — implicit scope
- — not limited to coroutines

Cons:

- — function has some restrictions since it is called from a destructor
- — depends on TLS state to detect success and fail, which may not be available on all platforms. Also, the detection can be confused when exceptions are transported or continuations resumed within the scope of an instance of the `scope_success` and `scope_fail` types.
- — there is no support for neither-a-result-nor-an-error and adding it would require adding more of the fragile TLS dependencies or a language feature.

4.2.3 scope_success, scope_fail, scope_done blocks

A language feature based on the `scope_guard` pattern would be another way to introduce support for fail/success/done interception.

bikeshedding aside..

Imagine that `break return` is a statement that returns from the current function with neither-a-result-nor-an-error.

Imagine that `scope_success`, `scope_fail` and `scope_done` were keywords that introduced statements that started an implicit scope (same rules as variable declarations) and introduced a block to run at the end of that scope. The `scope_..` blocks introduce a new scope within the current scope of the current function and can participate in the control flow of the current scope of the current function (using `goto`, `return`, `break return`, `break` and `continue`).

Finally, imagine that any type is allowed to implement `operator break return()`. `operator break return()` will be called when an object instance goes out of scope with neither-a-result-nor-an-error in flight.

Here is the example from the scope paper [P0052R10] with this proposal:

```
void grow(vector<int>& v){
  scope_success { cout << "Good!" << endl; };
  v.resize(1024);
}
```

Here is the example from `std::optional` in this paper with this proposal:

```
int op() {
  if (!has_feature()) {
    break return; // emits neither-a-result-nor-an-error
  }
  return feature();
}

void foo() {
  // ..
  {
    scope_done { cout << "feature unsupported!" << endl; };
    auto i = op();
    // use i..
  } // jumps here, when the feature is
    // not supported, runs scope_done
  // ..
} // jumps here, after running scope_done,
  // when the feature is not supported and
  // emits neither-a-result-nor-an-error

void bar() {
  // ..
  for(;;) {
    scope_done { break; };
    auto i = op();
    // use i..
  } // jumps here, when the feature is
    // not supported in op(), runs the
    // scope_done block which breaks
    // out of the for loop and runs
    // the rest of bar() normally.
    // bar() does not emit
    // neither-a-result-nor-an-error
  // ..
}
```

Pros:

- implicit scope
- not limited to coroutines
- safer than library solutions because the compiler+runtime owns the semantics
- no restrictions on the block contents since they are not run in the context of a destructor.

Cons:

- composition with existing functions that do not support neither-a-result-nor-an-error need the compiler+runtime to call ‘std::terminate()

5 Credits

This paper was influenced by hosts of people over decades.

- **Marc Barbour** and **Mark Lawrence** were fundamental to Kirk's first attempt to design more regular callbacks in a COM environment.
- **Aaron Lahman** was involved in that first attempt as well and introduced Kirk to the Reactive-Extensions libraries because he saw the similarity.
- **Erik Meijer** and his team took a very different path to arrive at a destination that resonated strongly with Kirk's goals
- *Microsoft Open Technologies Inc.* led by **Jean Paoli**, encouraged and supported Kirk's subsequent investment in finishing Aaron's C++ Rx prototype and then rewriting it to shift from interfaces to compile-time polymorphism.
- **Ben Christensen** drove changes to RxJava and his communication around those changes affected the design Kirk chose for rxcpp
- **Grigorii Chudnov**, **Valery Kopylov** and all the other amazing contributors to rxcpp over the years
- **Eric Niebler**, **Lee Howes** and **Lewis Baker** who more than anyone else contributed to the content of the motivation section of this paper
- **Lewis Baker**'s excellent `stop_source`/`stop_token` design in [P0660R9]
- *CppCon*, *CppNow*, *CppRussia* and *CERN* (and the people behind those including; **Jon Kalb**, **Bryce Adelstein-Lebach**, **Sergey Platonov**, **Axel Naumann**) for all the opportunities to communicate the vision for cancellation in C++
- **Gor Nishanov** for the excellent coroutines in C++20 and the shout-outs and support for rxcpp over the years.

5.0.1 Afterthought: converting undefined behaviour to defined behaviour

Something that has occurred only after imagining a language solution, is how language support for neither-a-result-nor-an-error would allow converting undefined-behaviour into defined-behaviour in a new and clean way. A method that could not return a value and should not throw an exception can use `break return` to return neither-a-result-nor-an-error. neither-a-result-nor-an-error can propagate up until handled without requiring any explicit code for neither-a-result-nor-an-error in the intermediate functions.

Some cooperation between compiler and runtime would be required to turn an unhandled neither-a-result-nor-an-error into `std::terminate()`. One example of an unhandled neither-a-result-nor-an-error would be when a calling function was compiled without support for neither-a-result-nor-an-error and a callee returned neither-a-result-nor-an-error. This case would need to result in `std::terminate()` and this would need to be enforced by the compiler+runtime of the callee not the caller. '

6 References

[N4771] Jonathan Wakely. 2018. Working Draft, C++ Extensions for Networking.
https://wg21.link/n4771

[P0052R10] Peter Sommerlad, Andrew L. Sandoval. 2019. Generic Scope Guard and RAII Wrapper for the Standard Library.
https://wg21.link/p0052r10

[P0660R4] Nicolai Josuttis, Herb Sutter, Anthony Williams. 2018. A Cooperatively Interruptible Joining Thread.
https://wg21.link/p0660r4

[P0660R9] Nicolai Josuttis, Lewis Baker, Billy O'Neal, Herb Sutter, Anthony Williams. 2019. Stop Token and Joining Thread.
https://wg21.link/p0660r9

[P1055R0] Kirk Shoop, Eric Niebler, Lee Howes. 2018. A Modest Executor Proposal.
https://wg21.link/p1055r0

[P1341R0] Lewis Baker. 2018. Unifying Asynchronous APIs in the Standard Library.
https://wg21.link/p1341r0

[P1371R0] Sergei Murzin, Michael Park, David Sankel, Dan Sarginson. 2019. Pattern Matching.
https://wg21.link/p1371r0