Cancellation is not an Error

Document #:	D1677R1
Date:	2019-08-06
Project:	Programming Language C++ SG1 Concurrency and Parallelism SG13 IO LEWG Library Evolution EWG Evolution

Reply-to:	Kirk Shoop <kirkshoop@fb.com>

Contents

1 Changelog

R1

- ☒ Use bibliography for Cologne paper references
- ☒ add stack-frame analogy for callbacks
- ☒ add `fiber_context` unwind mechanisms
- ☒ add section on non-cpp-exception
- ☒ add code to show `throw(...)` usages that break unwind by exception
- ☒ fix code examples (apply Lewis' feedback)
- ☒ add `f(g(h()))`, `generator` & `retry` examples
- ☒ switch the term neither-a-result-nor-an-error to serendipitous-success

2 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 (the mechanism to signal a request for cancellation is covered by `stop_source` in C++20). In this paper a cancelled result is described as an instance of serendipitous-success (Credits go to Lisa Lippincott for coining this term).

The ideas in this paper have proved to be exceedingly difficult to communicate. Each time this conversation is begun with a new person the same process of exploring the options to represent a cancelled result from a function is repeated.

It is usually easy to discard using an `optional<T>` return value. This ease is due to the noise it introduces, so we can skip the much harder task of explaining that the return value is a poor representation of a cancelled result.

We cannot avoid the hard task of explaining why something like a `cancelled_error` exception or `error_code` is a poor representation of a cancelled result, because it does not appear at first glance to introduce a lot of noise. This paper is focused on explaining why errors are a bad way to represent a cancelled result.

> 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.

3 Background

The `jthread` paper [P0660R9] and the `fiber_context` paper revisions [P0876R5] & [P0876R6] each describe exception based mechanisms for unwinding a stack in response to a cancellation request. These mechanisms have all been removed in later paper revisions due to issues with using exceptions to unwind the stack.

The `jthread` paper also defines `stop_source` and `stop_token`. 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 value or an error. This paper will explore how an async function will complete when it is stopped and why that is not an exception or an error.

The `jthread` and `fiber_token` use cases involve unwinding more conventional stack frames. This same functionality is also required in other realms. [P1745R0] describes how to add support for unwinding a graph of coroutine frames without using errors. [P1660R0] describes a solution for unwinding a graph of dependent tasks.

4 Motivation

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

4.1 stack unwinding

[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 `fiber_context` paper [P0876R5] defined `unwind_exception`, and after similar discussion in San Diego, [P0876R6] replaced `unwind_exception` with a 'platform exception' that did not run catch blocks. The 'platform exception' was removed after more discussion in Cologne.

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

4.1.1 C++ exception

4.1.1.1 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` exception 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 to be unwound
- 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.

4.1.1.2 transporting `std::interrupted`

`std::exception_ptr` and `std::current_exception()` were introduced to support async facilities like `std::thread`, 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`.

```
void f(MyCallback out) {
  try {
    out(g());
  }
  catch(...) {
    // should std::interrupted be forwarded to out or
    // should it be used to unwind f()?
    out.error(std::current_exception());
  }
}
```

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`?

4.1.2 non-cpp-exception

The issues of ignoring and transporting C++ exceptions for unwind has led to alternative designs that involve a non-cpp-exception.

A non-cpp-exception can have different interactions with catch blocks than C++ exceptions. Three of these potential interaction choices are explored here.

4.1.2.1 non-cpp-exception that does not execute catch blocks

One option is to never run any catch block during unwind. This depends on destructors to do all cleanup and restore all invariants.

Here is an example of code that would not be safe to have on the stack when catch blocks are not executed.

Example: unlock will not be called on unwind.

```
void f(Foo foo) {
  try {
    foo.lock();
    g();
    foo.unlock();
  }
  catch(...) {
    // invariant violation
    foo.unlock();
  }
}
```

4.1.2.2 non-cpp-exception that will execute catch(...) blocks normally

One option is to run each `catch(...)` block during unwind. This allows `catch(...)` blocks and destructors to cleanup and restore all invariants.

Here are examples of code that would not be safe to have on the stack when `catch(...)` blocks are run normally.

Example: The unwind will be stopped and the result of `std::current_exception()` would be used even though there is no valid value for it to return.

```
void f(Foo foo) {
  Bar* bar = nullptr;
  try {
    bar = g();
  }
  catch(...) {
    // what does current_exception() return for non-cpp-exception?
    // what do other exception related functions return?
    foo.error(std::current_exception());

    // is it ok for this to suppress the non-cpp-exception thus stopping the unwind?
    return;
  }
  foo(bar);
}
```

Example: The unwind would not run the catch and thus would not call unlock.

```
void f(Foo foo) {
  try {
    foo.lock();
    g();
    foo.unlock();
  }
  catch(const std::exception&) {
    // invariant violation
    foo.unlock();
  }
}
```

4.1.2.3 non-cpp-exception that will execute catch(...) blocks and force a rethrow at the end of the block

One option is to run each `catch(...)` block during unwind and then unconditionally rethrow at the end of the catch block. This allows `catch(...)` blocks and destructors to cleanup and restore invariants.

Here are examples of code that would not be safe to have on the stack when `catch(...)` blocks are run and then forced to rethrow.

Example: unlock will not be called on unwind.

```
void f(Foo foo) {
  foo.lock();
  try {
    g();
  }
  catch(...) {
  }
  // invariant violation
  foo.unlock();
}
```

Example: The result of `std::current_exception()` would be used even though there is no valid value for it to return. Control-flow choices are ignored.

```
void f(Foo foo) {
  Bar* bar = nullptr;
  try {
    bar = g();
  }
  catch(...) {
    // what does current_exception() return for non-cpp-exception?
    // what do other exception related functions return?
    foo.error(std::current_exception());

    // does not skip the rethrow
    return;
  }
  foo(bar);
}
```

Example: The unwind would not run the catch and thus would not call unlock.

```
void f(Foo foo) {
  try {
    foo.lock();
    g();
    foo.unlock();
  }
  catch(const std::exception&) {
    // invariant violation
    foo.unlock();
  }
}
```

4.2 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 value (and in this case, emits no error).
- the `when_all()` (aka `zip()`) algorithm which cancels the other producers when one completes with an error.
- the `take_until()` algorithm which cancels the source when thetrigger completes and cancels the trigger when the source completes.
- the `timeout()` algorithm which cancels the 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).

4.2.1 `when_any()`

One expression of the `when_any()` algorithm takes a set of async functions that have a common return type and returns the result of the first async function to complete with a value or error and cancels the rest and emits the value 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 value, with an error and with serendipitous-success. When all the async functions complete with serendipitous-success, then `when_any()` must complete with serendipitous-success.

NOTE: For the purpose of comparison this paper will use the Callback naming specified in [P1660R0] as an example of multiple function style. The names chosen for a particular expression of the multiple function style do not affect this proposal.

Table 1: generic `when_any()` algorithm example (simplified for clarity)

function

```
namespace when_any_alg {

template<class C>
struct result {
  C c_;
  atomic<int> remain_ = 2;
  function<void(C)> r_;
  stop_source stop_;
  void defer() {
    if (--remain_ == 0) {
      if (!r_) {
        c_.done();
      } else {
        r_(c_);
      }
    }
  }
};
```

pipe operator

```
namespace when_any_alg {

template<class C>
struct result {
  C c_;
  atomic<int> remain_ = 2;
  function<void(C)> r_;
  stop_source stop_;
  void defer() {
    if (--remain_ == 0) {
      if (!r_) {
        c_.done();
      } else {
        r_(c_);
      }
    }
  }
};
```

function	pipe operator

```
template<class R>
struct when_any_callback {
  R r_;
  void operator()(auto... vn) {
    if (stop_.request_stop()) {
      r_->r_ = [t=make_tuple(vn...)]
        (auto c) {apply(c, t);};
    }
    r_->defer();
  }
  void error(auto e) noexcept {
    if (stop_.request_stop()) {
      r_->r_ = [e]
        (auto c) {c.error(e);};
    }
    r_->defer();
  }
  void done() noexcept {
    r_->defer();
  }
};
```

```
template<class R>
struct when_any_callback {
  R r_;
  void operator()(auto... vn) {
    if (stop_.request_stop()) {
      r_->r_ = [t=make_tuple(vn...)]
        (auto c) {apply(c, t);};
    }
    r_->defer();
  }
  void error(auto e) noexcept {
    if (stop_.request_stop()) {
      r_->r_ = [e]
        (auto c) {c.error(e);};
    }
    r_->defer();
  }
  void done() noexcept {
    r_->defer();
  }
};
```

```
template<class S0, class S1>
struct when_any_sender {
  S0 s0_;
  S1 s1_;
  void submit(Callback auto c) {
    auto r = make_shared<
      result<decltype(c)>{c};
    s0_.submit(
      when_any_callback<decltype(r)>{
        r});
    s1_.submit(
      when_any_callback<decltype(r)>{
        r});
  }
};
```

```
template<class S0, class S1>
struct when_any_sender {
  S0 s0_;
  S1 s1_;
  void submit(Callback auto c) {
    auto r = make_shared<
      result<decltype(c)>{c};
    s0_.submit(
      when_any_callback<decltype(r)>{
        r});
    s1_.submit(
      when_any_callback<decltype(r)>{
        r});
  }
};
```

function	pipe operator

```
struct fn {
  auto operator()(
    Sender auto s0, Sender auto s1) {
    return when_any_sender<
      decltype(s0), decltype(s1)>{s0, s1};
  }
};

}
constexpr inline when_any_alg::fn when_any{};
```

```
template<class S1>
struct pipe_fn {
  S1 s1_;
  auto operator()(Sender auto s0) {
    return when_any_sender<
      decltype(s0), S1>{s0, s1_};
  }
};

struct fn {
  auto operator()(Sender auto s1) {
    return pipe_fn<decltype(s1)>{s1};
  }
};

}
constexpr inline when_any_alg::fn when_any{};
```

When some errors are supposed to terminate the operation early and others are not supposed to terminate the operation early, then an additional predicate is needed to select when to terminate early. If S0 calls `error()` with `cancelled_error` then S1 may still complete with a value. If S0 completes with `timeout_error` then it does not matter what S1 will complete with.

Table 2: generic `when_any()` algorithm example - `done()` vs. inspect `error()`

done()	inspect error()

```
namespace when_any_alg {



template<class C>
struct result {

  C c_;
  atomic<int> remain_ = 2;
  function<void(C)> r_;
  stop_source stop_;
  void defer() {
    if (--remain_ == 0) {
      if (!r_) {
        c_.done();
      } else {
        r_(c_);
      }
    }
  }
};
```

```
namespace when_any_alg {

struct when_any_stopped {
};
template<class P, class C>
struct result {
  P p_;
  C c_;
  atomic<int> remain_ = 2;
  function<void(C)> r_;
  stop_source stop_;
  void defer() {
    if (--remain_ == 0) {
      if (!r_) {
        c_.error(when_any_stopped{});
      } else {
        r_(c_);
      }
    }
  }
};
```

done()

```
template<class R>
struct when_any_callback {
  R r_;
  void operator()(auto... vn) {
    if (stop_.request_stop()) {
      r_->r_ = [t=make_tuple(vn...)]
        (auto c) {apply(c, t);};
    }
    r_->defer();
  }
  void error(auto e) noexcept {

    if (stop_.request_stop()) {
      r_->r_ = [e]
        (auto c) {c.error(e);};
    }

    r_->defer();
  }
  void done() noexcept {
    r_->defer();
  }
};
```

inspect error()

```
template<class R>
struct when_any_callback {
  R r_;
  void operator()(auto... vn) {
    if (stop_.request_stop()) {
      r_->r_ = [t=make_tuple(vn...)]
        (auto c) {apply(c, t);};
    }
    r_->defer();
  }
  void error(auto e) noexcept {
    if (r_->p_(e)) {
      if (stop_.request_stop()) {
        r_->r_ = [e]
          (auto c) {c.error(e);};
      }
    }
    r_->defer();
  }



};
```

```
template<class S0, class S1>
struct when_any_sender {

  S0 s0_;
  S1 s1_;
  void submit(Callback auto c) {
    auto r = make_shared<
      result<decltype(c)>{c};
    s0_.submit(
      when_any_callback<decltype(r)>{
        r});
    s1_.submit(
      when_any_callback<decltype(r)>{
        r});
  }
};
```

```
template<class P, class S0, class S1>
struct when_any_sender {
  P p_;
  S0 s0_;
  S1 s1_;
  void submit(Callback auto c) {
    auto r = make_shared<
      result<P, decltype(c)>{p_, c};
    s0_.submit(
      when_any_callback<decltype(r)>{
        r});
    s1_.submit(
      when_any_callback<decltype(r)>{
        r});
  }
};
```

done()

```
template<class S1>
struct pipe_fn {

  S1 s1_;
  auto operator()(Sender auto s0) {
    return when_any_sender<
      decltype(s0), S1>{s0, s1_};
  }
};

struct fn {

  auto operator()(Sender auto s1) {
    return pipe_fn<decltype(s1)>{s1};
  }
};

}
constexpr inline when_any_alg::fn when_any{};
```

inspect error()

```
template<class P, class S1>
struct pipe_fn {
  P p_;
  S1 s1_;
  auto operator()(Sender auto s0) {
    return when_any_sender<
      P, decltype(s0), S1>{p_, s0, s1_};
  }
};

struct fn {
  template<class P>
  auto operator()(P p, Sender auto s1) {
    return pipe_fn<P, decltype(s1)>{p, s1};
  }
};

}
constexpr inline when_any_alg::fn when_any{};
```

The additional complexity of the predicate impacts each algorithm that cancels. The complexity also impacts each use of algorithms that cancel.

A predicate that filters the errors would have some essential complexity and overhead.

```
struct should_fail {
  bool operator()(std::error_code e) {
   //..
  }
  bool operator()(std::exception_ptr e) {
    try {
      std::rethrow_exception(e);
    } // support for specific exceptions..
    catch(...) { // default to failure
      return true;
    }
    return false;
  }
  // support for other error types..
};
```

With `done()` the error filtering concern can be extracted from `when_any()` and the rest of the algorithms that cancel. A `filter_error()` algorithm would take the predicate and forward the `error()` if the predicate returned true.

Table 3: using `when_any` to compose async (`get_data`)

`done()`	`inspect error()`

`auto foo() {`
`  return get_data(server0) |`
`  filter_error(should_fail{}) |`
`  when_any(get_data(server1) |`
`  filter_error(should_fail{}));`
`}`
	`auto foo() {`
	`  return get_data(server0) |`
	`  when_any(should_fail{},`
	`  get_data(server1));`
	`}`

Observe that the `should_fail` predicate is unable to distinguish which source it is filtering when it is mixed into `when_any`. Separating the filter concern into `filter_error()` provides each source with a different filter.

Notice also, that introducing the `when_any_stopped` error type indicates that other algorithms will need to include that type in their predicates when they wish to filter it out.

When cancellation is not an error, algorithms that cancel are not concerned with errors and can pass through the value and error calls unchanged. The separation of concerns provided by the `when_any()` and `filter_error()` algorithms depend on cancellation not being an error.

4.3 Algorithms that respond to errors

One way to explore cancellation as separate from error is to show how treating cancellation as an error affects algorithms that respond to errors.

4.3.1 a generic `retry()` algorithm

`retry()` responds to errors by submitting the work again and again until it completes with success or is cancelled.

> NOTE: For the purpose of comparison this paper will use the Callback naming specified in [P1660R0] as an example of multiple function style. The names chosen for a particular expression of the multiple function style do not affect this proposal.

When `done()` and `error()` are separate, the code for `retry()` does not need to inspect errors.

Table 4: generic `retry()` algorithm example (simplified for clarity)

function

```
namespace retry_alg {

template<class S, class C>
struct retry_callback {
  S s_;
  C c_;
  void operator()(auto... vn) {
    c_(vn...);
  }
  void error(auto) noexcept {
    s_.submit(*this);
  }
  void done() noexcept {
    c_.done();
  }
};

template<class S>
struct retry_sender {
  S s_;
  void submit(Callback auto c) {
    s_.submit(
      retry_callback<S, decltype(c)>{s_, c});
  }
};

struct fn {
  auto operator()(Sender auto s) {
    return retry_sender<decltype(s)>{s};
  }
};

}
constexpr inline retry_alg::fn retry{};
```

pipe operator

```
namespace retry_alg {

template<class S, class C>
struct retry_callback {
  S s_;
  C c_;
  void operator()(auto... vn) {
    c_(vn...);
  }
  void error(auto) noexcept {
    s_.submit(*this);
  }
  void done() noexcept {
    c_.done();
  }
};

template<class S>
struct retry_sender {
  S s_;
  void submit(Callback auto c) {
    s_.submit(
      retry_callback<S, decltype(c)>{s_, c});
  }
};

struct pipe_fn {
  auto operator()(Sender auto s) {
    return retry_sender<decltype(s)>{s};
  }
};

struct fn {
  auto operator()() {
    return pipe_fn{};
  }
};

}
constexpr inline retry_alg::fn retry{};
```

When some errors are supposed to retry and others are not supposed to retry then an additional predicate is needed to select when to retry.

Table 5: generic `retry()` algorithm example with `done()` vs. inspecting `error()`

`done()`

```
namespace retry_alg {

template<class S, class C>
struct retry_callback {
  S s_;
  C c_;
  void operator()(auto... vn) {
    c_(vn...);
  }
  void error(auto) noexcept {
    s_.submit(*this);
  }
  void done() noexcept {
    c_.done();
  }
};

template<class S>
struct retry_sender {
  S s_;
  void submit(Callback auto c) {
    s_.submit(
      retry_callback<S, decltype(c)>{s_, c});
  }
};

struct pipe_fn {
  auto operator()(Sender auto s) {
    return retry_sender<decltype(s)>{s};
  }
};

struct fn {
  auto operator()() {
    return pipe_fn{};
  }
};

}
constexpr inline retry_alg::fn retry{};
```

inspect `error()`

```
namespace retry_alg {

template<class P, class S, class C>
struct retry_callback {
  P p_;
  S s_;
  C c_;
  void operator()(auto... vn) {
    c_(vn...);
  }
  void error(auto e) noexcept {
    if (p_(e)) {
      s_.submit(*this);
    } else {
      c_.error(e);
    }
  }
};

template<class P, class S>
struct retry_sender {
  P p_;
  S s_;
  void submit(Callback auto c) {
    s_.submit(
      retry_callback<P, S, decltype(c)>{
        p_, s_, c});
  }
};

template<class P>
struct pipe_fn {
  P p_;
  auto operator()(Sender auto s) {
    return retry_sender<P, decltype(s)>{
      p_, s};
  }
};

struct fn {
  template<class P>
  auto operator()(P p) {
    return pipe_fn<P>{p};
  }
};

}
constexpr inline retry_alg::fn retry{};
```

done()	inspect error()

The additional complexity of the predicate impacts each algorithm that responds to errors. The complexity also impacts each use of algorithms that respond to errors.

A predicate that filters the errors would have some essential complexity and overhead.

```
struct should_retry {
  bool operator()(std::error_code e) {
   //..
  }
  bool operator()(std::exception_ptr e) {
    try {
      std::rethrow_exception(e);
    } // support for specific exceptions..
    catch(...) { // default to retry
      return true;
    }
    return false;
  }
  // support for other error types..
};
```

With `done()` the error filtering concern can be extracted from `retry()` and the rest of the algorithms that respond to errors. A `filter_error()` algorithm would take the predicate and forward the `error()` if the predicate returned true and emit `done()` when the predicate returned false.

Table 6: using `retry` to compose async (`get_data`)

done()	inspect error()

`auto foo() {`
`  return get_data() |`
`    filter_error(should_retry{}) |`
`    retry();`
`}`
	`auto foo() {`
	`  return get_data() |`
	`    retry(should_retry{});`
	`}`

When cancellation is not an error, algorithms that respond to errors are only concerned with errors and can pass through the value and done calls unchanged. The separation of concerns provided by the `retry()` and `filter_error()` algorithms depend on cancellation not being an error.

4.4 Callbacks

As the most common pattern for expressing async, callbacks also need to be called with serendipitous-success. There is a lot to be said about callbacks and ([P1678R0], latest) 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 a serendipitous-success.

4.4.1 destructor style

There are reasons not to use the destructor to signal that there was serendipitous-success.

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 serendipitous-success 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 serendipitous-success 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 serendipitous-success signal. But there is another, async vs object lifetime.

4.4.2 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.

4.4.3 `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::tuples`, but still worse than when it is an plain argument to the function. For instance, something that transforms the value from one type to another has to check the error, unpack the value or error and repack the transformed value or original error into the outgoing expected type.

4.4.4 multiple function style

Some of the tradeoffs encountered when mixing errors and values into the same 'channel' (where function arguments and function return values 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 value is equivalent to the separation of `return value` and `throw/catch` in the language. Using multiple functions for error and value produces very different tradeoffs than when mixing

error and value together in one function. The [`std::promise` type:] is an example of using multiple functions for error and value 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 value. 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.
— [P1660R0] defines the Callback concept that subsumes the Invocable and Fallback concepts resulting in the names `Invocable::operator()(Tn...)`, `Fallback::error(E)` and `Fallback::done()`. [P1660R0] includes an example implementation.

The Callback concept defined in [P1660R0] 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 serendipitous-success

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<>...>`

4.4.5 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

4.5 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.

4.6 sync functions (not a typo)

sync functions also need to complete with serendipitous-success.

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

4.6.1 coroutine generator

This example is also made clearer by avoiding Iterators.

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

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

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

// assumes that g.next()
// completes with serendipitous-success
void foo() {
  auto g = fortyTwos();
  for(;;) {
    auto fortyTwo = g.next();
  }
}
```

Notice that `generator` is a channel that connects two loops, the producer loop and the consumer loop. Each loop may independently break to exit. When break is used in one loop, the channel must have a way to cause the other loop to break as well. This signal is not an error, `throw` is not a good way to represent `break`. The iterator concept encodes `break` into the value by allowing `end()` to represent an empty iterator.

Another way to say the same thing, is that types like `generator<int>` actually create a coroutine whose body is allowed to produce two different value types. `co_yield 42;` resolves the matching `g.next()` with an int while `co_return;` resumes the matching `g.next()` with void.

Obviously `next()` returning `int` and `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 iterators that are either a proxy to the yielded value or compare equal to `end()` when `void` is returned.

While this value encoding seems natural for Range, it is not so palatable for `std::optional`.

4.6.2 `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 return value rather than as an exception. This path was rejected previously because of the impact that it would have on code. all return values 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 value or forward on the empty `std::optional`. 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 serendipitous-success.

4.7 Examples

An imaginary world, where a sync function can complete with serendipitous-success, would have cleaner code.

Table 7: `op()` that produces an empty value when a feature is not supported vs. imaginary `op()` that emits serendipitous-success

Real

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

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

Imaginary

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

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

Table 8: `op()` that produces an error when a feature is not supported vs. imaginary `op()` that emits serendipitous-success

Real

```
int op() {
  if (!has_feature()) {
    throw unsupported_error();
  }
  return feature();
}

void foo() {
  // ..
  try {
    auto i = op();
    // use *i..
  } catch(const unsupported_error&) {
    return;
  }
} // jumps here when the feature is
  // not supported
```

Imaginary

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

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

4.7.1 `f(g(h()))`

Given a simple composition of the functions `f()`, `g()`, `h()`, demonstrate various forms of cancellation within their implementations.

— `h()` will return a task that produces a void result each time it is invoked (unless it has been cancelled)
— `g()` will return a task that that calls the task argument 9 times then cancels
— `f()` will wait for the task argument to complete and print the results

These first implementations are sync functions that use `optional<>` to compose `f()`, `g()`, and `h()`.

Table 9: composition - explicit vs. lazy

Explicit

```
// usage
int main() {
  f();
}
```

```
using void_value = tuple<>;
auto h(stop_token stop)
  -> optional<void_value> {
  if(stop.stop_requested()) {
    return nullopt;
  }
  return void_value{};
}
```

```
auto g(stop_token stop)
  -> optional<int> {
  int count = 0;
  stop_source stopInner;
  for (;;) {
    if (stop.stop_requested()) {
      stopInner.request_stop();
    }
    if (!h(stopInner.get_token())) {
      break;
    }
    if (++count >= 9) {
      stopInner.request_stop();
    }
  }
  return count;
}
```

Lazy

```
// usage
int main() {
  f(g(h()));
}
```

```
using void_value = tuple<>;
auto h() {
  return [](stop_token stop)
    -> optional<void_value> {
    if(stop.stop_requested()) {
      return nullopt;
    }
    return void_value{};
  };
}
```

```
auto g(auto h) {
  return [h](stop_token stop)
    -> optional<int> {
    int count = 0;
    stop_source stopInner;
    for (;;) {
      if (stop.stop_requested()) {
        stopInner.request_stop();
      }
      if (!h(stopInner.get_token())) {
        break;
      }
      if (++count >= 9) {
        stopInner.request_stop();
      }
    }
    return count;
  };
}
```

Explicit

```
void f() {
  exception_ptr ex;
  stop_source stop;
  optional<int> count;
  thread t{[&](){
    this_thread::sleep_for(100ms);
    stop.request_stop();
  }};
  try {
    count = g(stop.get_token());
  } catch(...) {
    ex = current_exception();
  }
  auto w = (
    !!count ? "completed" :
    !!ex ? "failed" :
    !count ? "stopped" : "invalid");
  printf("which %s, count %d", w, *count);
  t.join();
}
```

Lazy

```
void f(auto g) {
  exception_ptr ex;
  stop_source stop;
  optional<int> count;
  thread t{[&](){
    this_thread::sleep_for(100ms);
    stop.request_stop();
  }};
  try {
    count = g(stop.get_token());
  } catch(...) {
    ex = current_exception();
  }
  auto w = (
    !!count ? "completed" :
    !!ex ? "failed" :
    !count ? "stopped" : "invalid");
  printf("which %s, count %d", w, *count);
  t.join();
}
```

The Explicit composition demonstrates one level of composition, Lazy demonstrates another level of composition that is needed to allow functions to be chained in an expression. This is similar to the difference between `std::transform` and `std::views::transform`.

The rest of these implementations have been structured to support the Lazy form of composition.

This demonstrates an async implementation with a Library composition model that has cancellation support.

> NOTE: For the purpose of comparison this paper will use the Callback naming specified in [P1660R0] as an example of multiple function style. The names chosen for a particular expression of the multiple function style do not affect this proposal.

```
struct h_task {
  void submit(Callback auto c) {
    try {
      thread t([c]() mutable {
        if(c.get_stop_token().stop_requested()) {
          c.done();
        } else {
          c();
        }
      });
      t.detach();
    } catch(...) {
      c.error(std::current_exception());
    }
  }
};
h_task h() {
```

```
  return {};
}
```

```
template<class H, class C>
struct g_callback {
  H h_;
  C c_;
  int count_ = 0;
  stop_source stop_;
  stop_token get_stop_token() {
    return stop_.get_token();
  }
  void operator()() {
    if (c_.get_stop_token().stop_requested()) {
      stop_.request_stop();
    }
    ++count_;
    try {
      if (count_ < 9) {
        h_.submit(*this);
      } else {
        stop_.request_stop();
        h_.submit(*this);
      }
    } catch(...) {
      c_.error(std::current_exception());
    }
  }
  void error(auto e) noexcept {
    if (c_.get_stop_token().stop_requested()) {
      stop_.request_stop();
    }
    c_.error(e);
  }
  void done() noexcept {
    if (c_.get_stop_token().stop_requested()) {
      c_.done();
      return;
    }
    try {
      c_(count_);
    } catch(...) {
      c_.error(std::current_exception());
    }
  }
};
template<class H>
struct g_task {
  H h_;
  void submit(Callback auto c) {
    try {
      h_.submit(g_callback<H, decltype(c)>{
        h_, c});
    } catch(...) {
```

```
      c.error(std::current_exception());
    }
  }
};
auto g(Sender auto h)
  -> g_task<decltype(h)> {
  return {h};
}
```

```
struct f_callback {
  stop_token stop_;
  atomic<int>& which_;
  atomic<int>& count_;
  stop_token get_stop_token() {
    return stop_;
  }
  void operator()(int count) {
    count_.exchange(count);
    which_.exchange(1);
  }
  void error(auto e) noexcept {
    which_.exchange(2);
  }
  void done() noexcept {
    which_.exchange(3);
  }
};
void f(auto g) {
  stop_source stop;
  atomic<int> which{0};
  atomic<int> count{0};
  thread t{[&](){
    this_thread::sleep_for(100ms);
    stop.request_stop();
  }};
  f_callback r{stop.get_token(), which, count};
  g.submit(r);
  stop.request_stop();
  while(which.load() == 0);
  auto w = (
    which.load() == 1 ? "completed" :
    which.load() == 2 ? "failed" :
    which.load() == 3 ? "stopped" : "invalid");
  printf("which %s, count %d", w, count.load());
  t.join();
}
```

This implementation demonstrates a sync implementation with a Library composition model that uses the return value to support cancellation.

Table 10: serendipitous-success - return value vs. imaginary language feature

Real

```
using void_value = tuple<>;
auto h() {
  return [](stop_token stop)
    -> optional<void_value> {
    if(stop.stop_requested()) {
      return nullopt;
    }
    return void_value{};
  };
}
```

```
auto g(auto h) {
  return [h](stop_token stop)
    -> optional<int> {
    int count = 0;
    stop_source stopInner;
    for (;;) {
      if (stop.stop_requested()) {
        stopInner.request_stop();
      }
      if (!h(stopInner.get_token())) {
        break;
      }
      if (++count >= 9) {
        stopInner.request_stop();
      }
    }
    return count;
  };
}
```

Imaginary

```
auto h() {
  return [](stop_token stop) {
    if(stop.stop_requested()) {
      break return;
    }
    return;
  };
}
```

```
auto g(auto h) {
  return [h](stop_token stop)
    -> int {
    int count = 0;
    stop_source stopInner;
    for (;;) {
      if (stop.stop_requested()) {
        stopInner.request_stop();
      }
      {
        scope_done {break;}
        h(stopInner.get_token());
      }
      if (++count >= 9) {
        stopInner.request_stop();
      }
    }
    return count;
  };
}
```

Real	Imaginary

```
void f(auto g) {
  exception_ptr ex;
  stop_source stop;
  optional<int> count;
  thread t{[&](){
    this_thread::sleep_for(100ms);
    stop.request_stop();
  }};
  try {
    count = g(stop.get_token());
  } catch(...) {
    ex = current_exception();
  }
  auto w = (
    !!count ? "completed" :
    !!ex ? "failed" :
    !count ? "stopped" : "invalid");
  printf("which %s, count %d", w, *count);
  t.join();
}
```

```
void f(auto g) {
  exception_ptr ex;
  stop_source stop;
  optional<int> count;
  thread t{[&](){
    this_thread::sleep_for(100ms);
    stop.request_stop();
  }};
  auto print = [&]() {
    auto w = (
      !!count ? "completed" :
      !!ex ? "failed" :
      !count ? "stopped" : "invalid");
    printf("which %s, count %d", w, *count);
  };
  try {
    scope_done {print();}
    count = g(stop.get_token());
  } catch(...) {
    ex = current_exception();
  }
  print();
  t.join();
}
```

This implementation demonstrates a sync implementation with a Library composition model that uses `throw` to support cancellation.

Table 11: serendipitous-success - `throw` vs. imaginary language feature

Real	Imaginary

```
struct stopped_exception : exception {};
auto h() {
  return [](stop_token stop) {
    if(stop.stop_requested()) {
      throw stopped_exception{};
    }
    return ;
  };
}
```

```
auto h() {
  return [](stop_token stop) {
    if(stop.stop_requested()) {
      break return;
    }
    return;
  };
}
```

Real	Imaginary

```
auto g(auto h) {
  return [h](stop_token stop)
    -> int {
    int count = 0;
    stop_source stopInner;
    for (;;) {
      if (stop.stop_requested()) {
        stopInner.request_stop();
      }
      try {
        h(stopInner.get_token());
      } catch (const stopped_exception&) {
        break;
      }
      if (++count >= 9) {
        stopInner.request_stop();
      }
    }
    return count;
  };
}
```

```
auto g(auto h) {
  return [h](stop_token stop)
    -> int {
    int count = 0;
    stop_source stopInner;
    for (;;) {
      if (stop.stop_requested()) {
        stopInner.request_stop();
      }
      {
        scope_done {break;}
        h(stopInner.get_token());
      }
      if (++count >= 9) {
        stopInner.request_stop();
      }
    }
    return count;
  };
}
```

```
void f(auto g) {
  exception_ptr ex;
  stop_source stop;
  optional<int> count;
  thread t{[&](){
    this_thread::sleep_for(100ms);
    stop.request_stop();
  }};
  try {
    count = g(stop.get_token());
  } catch (const stopped_exception&) {
  } catch(...) {
    ex = current_exception();
  }
  auto w = (
    !!count ? "completed" :
    !!ex ? "failed" :
    !count ? "stopped" : "invalid");
  printf("which %s, count %d", w, *count);
  t.join();
}
```

```
void f(auto g) {
  exception_ptr ex;
  stop_source stop;
  optional<int> count;
  thread t{[&](){
    this_thread::sleep_for(100ms);
    stop.request_stop();
  }};
  auto print = [&]() {
    auto w = (
      !!count ? "completed" :
      !!ex ? "failed" :
      !count ? "stopped" : "invalid");
    printf("which %s, count %d", w, *count);
  };
  try {
    scope_done {print();}
    count = g(stop.get_token());
  } catch(...) {
    ex = current_exception();
  }
  print();
  t.join();
}
```

5 Conclusions

The cancellations, covered in Motivation above, are not errors and the functions that were cancelled should complete with serendipitous-success.

Further, cancellation is not the only case covered in Motivation above, where a function would benefit from completing with serendipitous-success.

Finally, serendipitous-success is a signal that does not have a good representation using the existing forms of function output.

5.1 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.

5.1.1 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)

5.1.2 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`.

5.1.3 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 value.
— `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.

5.2 Contrast function-taking-a-callback with function

— A function-taking-a-callback is invoked from a stack frame that may not exist when return-value|exception is emitted

— The only remaining fragment of the stack frame that invoked the function-taking-a-callback is the callback argument

— The signals return-value|exception that would be delivered to the stack frame that invoked a function-taking-a-callback must be delivered to the callback argument

6 Proposals

There are designs that can support value and error and serendipitous-success for both library and language.

6.1 Library

When adding async functions to the library there must be a way to represent a value and an error and serendipitous-success.

Currently the ways to represent value and error were covered in `std::optional`, `f(g(h()))`, coroutine generator, Callbacks and stack unwinding above. Of these, the only one with a working solution for a value and an error and serendipitous-success is the multiple function style in Callbacks. The `async_accept_completion` example is 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 serendipitous-success

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<>>`

6.2 Language

As noted in `std::optional` above, there is no language feature that supports serendipitous-success. Here are some thoughts on what this might look like in the language.

6.2.1 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

6.2.2 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 serendipitous-success and adding it would require adding more of the fragile TLS dependencies or a language feature.

6.2.3 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 serendipitous-success.

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 serendipitous-success 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 serendipitous-success
  }
  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 and
  // emits serendipitous-success

void bar() {
  // ..
  scope_done { return; };
  auto i = op();
  // use i..
} // jumps here, when the feature is
  // not supported in op(), runs the
  // scope_done block which uses return
  // to exit bar() normally. bar() does
  // not emit serendipitous-success
```

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 serendipitous-success need the compiler+runtime to call ‘std::terminate()

6.2.4 Afterthought: converting undefined behaviour to defined behaviour

Something that has occurred only after imagining a language solution, is how language support for serendipitous-success 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 serendipitous-success. serendipitous-success can propagate up until handled without requiring any explicit code for serendipitous-success in the intermediate functions.

Some cooperation between compiler and runtime would be required to turn an unhandled serendipitous-success

into `std::terminate()`. One example of an unhandled serendipitous-success would be when a calling function was compiled without support for serendipitous-success and a callee returned serendipitous-success. 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.

7 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.
— **Lisa Lippincott** for coining 'serendipitous-success' to explain what the result of cancellation is rather than what it is not.

‘

8 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

[P0876R5] Oliver Kowalke, Nat Goodspeed. 2019. fiber_context - fibers without scheduler.
https://wg21.link/p0876r5

[P0876R6] Oliver Kowalke, Nat Goodspeed. 2019. fiber_context - fibers without scheduler.
https://wg21.link/p0876r6

[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

[P1660R0] Jared Hoberock, Michael Garland, Bryce Adelstein Lelbach, Michał Dominiak, Eric Niebler, Kirk Shoop, Lewis Baker, Lee Howes, David S. Hollman, Gordon Brown. 2019. A Compromise Executor Design Sketch.
https://wg21.link/p1660r0

[P1678R0] Kirk Shoop. 2019. Callbacks and Composition.
https://wg21.link/p1678r0

[P1745R0] Lewis Baker. 2019. Coroutine changes for C++20 and beyond.
https://wg21.link/p1745r0