Document Number: P0114R0
Date: 2015-09-25
Revises: N4453
Reply To: Christopher Kohlhoff <chris@kohlhoff.com>

Resumable Expressions (revision 1)

1 Introduction

This paper proposes *resumable expressions*, a pure language extension for coroutines with the following features:

- **Lightweight, efficient, "stackless" coroutines.** Memory use is exactly what is required. There is no need for reserved stack space.
- **Cheaply composable.** Composition within a resumable expression can be as cheap as a function call, and includes the opportunity for inlining.
- **No hidden memory allocations.** The memory representation of a resumable expression can be wherever you need it: on the stack, a global, or a member of a possibly heap-allocated object.
- **Only one new keyword.** Inspired by `constexpr`, a new `resumable` keyword that: 1) introduces a resumable expression; 2) marks a resumable function; and 3) when following the `break` keyword, defines a suspension point.

And perhaps most importantly:

- **Avoids viral flow control keywords.** Keywords like `await` and `yield` are not required. Coroutines are structured like normal code, and suspend *down*, which means that as a consequence the language extension...
- **Enables code reuse** between stackless and stackful coroutines, and normal threads.
- **Provides a building block** for implementing resumable flow control patterns, including (but not limited to) `await` and `yield`.

A resumable function looks like this:

```
resumable void print_1_to(int n)
{
  for (int i = 1;;)
  {
    std::cout << i << std::endl;
    if (++i > n) break;
    break resumable;
  }
}
```

Indicates that the function is resumable

Suspends execution of the resumable function at this point

and is used in a resumable expression like this:

```
resumable auto r = print_1_to(5);
while (!r.ready())
{
  std::cout << "resuming ... ";
  r.resume();
}
```

Declares a *resumable object* named `r` to evaluate the *resumable expression* `print_1_to(5)`

The `ready` function returns false until the resumable expression completes normally

No code associated with the resumable expression executes until we call `resume()`

to output:

```
resuming ... 1
resuming ... 2
resuming ... 3
resuming ... 4
resuming ... 5
```

1.1 Changes in this revision

This revision adds additional discussion on how certain design choices in N4402's up-and-out model impact performance and introduce the potential for unfairness and starvation. The revision also introduces some possible alternate syntax for the creation and suspension of resumable expressions.

2 Background

In Urbana, the committee discussed the alternative coroutines models put before it, namely *stackless* coroutines and *stackful* coroutines.

One outcome of this discussion was an acceptance that each model addresses different use cases, even if there is some overlap. Supporting these distinct uses is desirable, and as such both models remain on the table.

Another outcome was a determination that the two models were different enough that it was not worth trying to reconcile their syntax. Whereas stackful coroutines lend themselves to a pure library implementation, both of the stackless proposals on offer introduced new control flow keywords: `await` and `yield`.

Even so, a vocal minority maintained that the two models are not so fundamentally different as to require a wholly different syntax. As we shall see below, introducing a new syntax, purely for stackless coroutines, comes with significant costs in terms of maintainability and usability.

3 Introducing resumable expressions

In this paper we will introduce a new stackless coroutine proposal that addresses these limitations: *resumable expressions*. Modelled on `constexpr`, resumable expressions allow us to develop lightweight resumable functions without peppering our code with special-purpose keywords.

For example, when composing asynchronous operations it is no longer necessary to mark each asynchronous call with the keyword `await`.

Example using N4402 resumable functions

```
std::future<void> tcp_reader(int total)
{
  char buf[64 * 1024];
  auto conn =
    await Tcp::Connect("127.0.0.1", 1337);
  do
  {
    auto bytesRead =
      await conn.read(buf, sizeof(buf));
    total -= bytesRead;
  }
  while (total > 0);
}
```

Equivalent using resumable expressions

```
resumable void tcp_reader(int total)
{
  char buf[64 * 1024];
  auto conn =
    Tcp::Connect("127.0.0.1", 1337);
  do
  {
    auto bytesRead =
      conn.read(buf, sizeof(buf));
    total -= bytesRead;
  }
  while (total > 0);
}
```

In the resumable expressions version of this example, the asynchronous operations `Tcp::Connect()` and `conn.read()` are resumable functions. This means that we can call them directly, and efficiently, just as though they were normal functions. There is no need to use a `future` object to await their completion. Likewise, a resumable caller may incorporate our `tcp_reader()` function into a larger asynchronous composition with the same cost as a normal function call.

Furthermore, since `Tcp::Connect()` and `conn.read()` are called like normal functions, they **can** be normal functions. We have the flexibility to introduce (or remove) resumable functions further down the call tree, without the maintenance burden of a viral `await` keyword.

Similarly, when writing generators, we no longer need to use the `yield` keyword to generate each value. The equivalent semantics can be obtained using normal functions or function objects.

Example using N4402 resumable functions

```
generator<int> fib(int n)
{



  int a = 0;
  int b = 1;
  while (n-- > 0)
  {
    yield a;
    auto next = a + b;
    a = b;
    b = next;
  }


}
```

Equivalent using resumable expressions

```
generator<int> fib(int n)
{
  return {
    [=](auto yield) mutable
    {
      int a = 0;
      int b = 1;
      while (n-- > 0)
      {
        yield(a);
        auto next = a + b;
        a = b;
        b = next;
      }
    }
  };
}
```

However, when using resumable expressions we are **not required to use a type-erased container** to represent the resumable function object.

Example using N4402 resumable functions

```
generator<int> fib(int n)
{
  int a = 0;
  int b = 1;
  while (n-- > 0)
  {
    yield a;
    auto next = a + b;
    a = b;
    b = next;
  }
}
```

Equivalent using resumable expressions

```
template <class OutputIterator>
void fib(int n, OutputIterator out)
{
  int a = 0;
  int b = 1;
  while (n-- > 0)
  {
    *out++ = a;
    auto next = a + b;
    a = b;
    b = next;
  }
}
```

Notice anything? That's right: there is nothing specifically resumable about the version on the right. It is a plain ol' function template, and it works equally well when called as either a resumable function or as a regular function. However, there is no magic here: we are simply delegating suspension of the resumable function *down* to the output iterator's assignment operator.

But isn't this the downward suspension we already get with stackful coroutines? The difference is that, being stackless, resumable expressions are truly lightweight. Memory usage is exactly that required to represent the resumable function's maximum stack depth, and no more.

With nothing more than compiler support, resumable expressions provide a set of primitives on which we can build rich library abstractions for lightweight and efficient coroutines. And, perhaps more importantly, resumable expressions let us share library abstractions between stackless coroutines, stackful coroutines, and normal thread-based code.

4 Suspension models for coroutines and resumable functions

The key characteristic of a coroutine or resumable function is that it is able to suspend its own execution. Control can pass out of a function body in such a way that it can be later resumed at the same point. N4232 *Stackful Coroutines and Stackless Resumable Functions* describes two models of function suspension: *down* and *up-and-out*.

The *down* model is used by stackful coroutines, such as the Boost.Coroutine library and N3985 *A proposal to add coroutines to the C++ standard library*. To suspend, a resumable function simply performs an ordinary function call to what looks like an ordinary library function. The mechanics of suspension are delegated to this function.

```
void down_foo()
{
  // ... do some work ...
  suspend(); // suspend execution without returning from down_foo()
  // ... resume here and do more work ...
}
```

The *up-and-out* model is used by stackless coroutines, such as N4402 *Resumable Functions*, and N4244 *Resumable Lambdas*. In this model a resumable function suspends and returns control to the caller. This is achieved with the help of new keywords, such as `await` and `yield`, which act like a non-terminating return.

```
generator<int> up_and_out_foo()
{
  // ... do some work ...
  yield 42; // suspend execution and return 42
  // ... resume here and do more work ...
}
```

4.1 Up-and-out goes viral

These two suspension models have significantly different implications when we try to compose them in a call stack.

In the *down* model, stack composition is trivial. There is no requirement that the `suspend()` library function be called directly. We are able to interpose other functions into the call stack between our resumable function and the suspension point.

```
void down_bar()
{
  // ... do some work ...
  down_foo(); // suspension occurs inside down_foo()
  // ... do more work here after returning from down_foo() ...
}
```

With *up-and-out*, however, all functions in the stack must be explicitly marked as resumable. A caller suspends its own execution before invoking a callee. Only after the callee returns, and is in its terminal state, will the caller continue. This control flow is automatically managed by the new keywords, `await` and `yield`.

```
generator<int> up_and_out_bar()
{
  // ... do some work ...
  int a = await up_and_out_foo(); // suspend until up_and_out_foo() completes
  // ... resume here and do more work ...
}
```

As a consequence, *up-and-out* resumable functions have a viral impact on codebases. When we discover that, deep in a call stack, we have a newfound need to suspend execution, we

must change all calls in that stack to be resumable. Experiences with other languages that take a similar approach (such as Python) have shown that this quickly becomes a very costly maintenance nightmare.

Conversely, whenever we make a call to a resumable function we must remember to annotate the call with an `await` keyword. Failure to do so can result in hard to diagnose bugs due to the inadvertent addition of concurrency. For example:

```
std::future<void> tcp_writer(int total)
{
  auto conn = await Tcp::Connect("127.0.0.1", 1337);
  for (;;)
  {
    std::vector<char> data = generator_data();
    conn.write(std::move(data));
    update_connection_statistics();
  }
}
```

Oops, `write` is asynchronous but we forgot to put `await` on this line

This code is an attempt to implement a business requirement that states that, after every `write()`, we update some data via a call to `update_connection_statistics()`. The requirement also stipulates that `update_connection_statistics()` not be called until the side effects of the preceding write operation have been established.

Most of the time this implementation will work successfully, as the write operation completes immediately. Occasionally, however, TCP flow control will delay the write, and consequently the `tcp_writer` function will start a call to `update_connection_statistics()` before the write completes.

Resumable functions using the *down* model do not suffer from this issue. Under the *down* model, from the outside our connection's `write()` looks and behaves like a normal function:

```
class Tcp
{
public:
  // ...
  void write(std::vector<char> data);
  // ...
};
```

That is, it does not return control to its caller until it has established its side effects. Consequently, under the *down* model, we can implement our requirement without the accidental introduction of concurrency:

```
void tcp_writer(int total)
{
  auto conn = Tcp::Connect("127.0.0.1", 1337);
  for (;;)
  {
    std::vector<char> data = generator_data();
    conn.write(std::move(data));
    update_connection_statistics();
  }
}
```

No `await` keyword required on this line

Cannot get to this line until `write()` is complete

4.2 Islands of abstraction

Even though N4402's *up-and-out* resumable functions require new keywords, the overall control flow is still the same imperative approach found elsewhere in the language. Thus, as resumable functions pervade a codebase, developers will naturally wish to employ reusable abstractions for coordinating them. Unfortunately, we come unstuck as soon as we try to use the standard algorithms.

```
std::future<void> tcp_sender(std::vector<std:string> msgs)
{
  auto conn = await Tcp::Connect("127.0.0.1”, 1337);
  std::for_each(msgs.begin(), msgs.end(),
    [&conn](const std::string& msg)
    {
      await conn.write(msg.data(), msg.size());
    });
}
```

Not going to work

We are forced to develop alternative algorithms that know about `await`.

```
template <class I, class F>
std::future<void> resumable_for_each(I begin, I end, F f)
{
  for (I iter = begin; iter != end; ++iter)
    await f(*iter);
}

std::future<void> tcp_sender(std::vector<std:string> msgs)
{
  auto conn = await Tcp::Connect("127.0.0.1”, 1337);
  resumable_for_each(msgs.begin(), msgs.end(),
    [&conn](const std::string& msg)
    {
      await conn.write(msg.data(), msg.size());
    });
}
```

Over time we may end up with a complete set of `await`-enabled algorithms that mirror the existing standard algorithms. These algorithms will be independent and not interoperable with the existing algorithms. They are islands of abstraction.

4.3 Type erasure and fine grained polymorphism

Though not an inherent requirement of the up-and-out model, N4402 resumable functions introduce type erasure. This means that when we write a resumable function:

```
generator<int> foo()
{
  std::Vector<int> vec = { 1, 2, 3, 4 };
  for (auto i : vec)
    yield i;
}
```

the compiler generates some unnamed implementation type for us, representing the “stack frame” of the resumable function:

```
class __foo
{
  std::vector<int> vec;
  int i
  void __resume()
  {
    ...
  }
  ...
};
```

However, the type `__foo` is not accessible; our ability to manipulate the resumable function is limited to a type-erased wrapper: `coroutine_handle<void>`. Thus, every call to `coroutine_handle<void>::resume()` necessarily incurs the cost of an indirect function call. Performance is further impacted as the indirect call inhibits inlining.

In this example, our type-erased implementation is owned by a polymorphic wrapper, `generator<int>`. In the N4499 Draft Wording for Coroutines, `std::example::generator<>` is

shown (in example form only) to rely on `coroutine_handle<>` to provide type erasure. Rather than supporting arbitrary types satisfying some `Generator` type requirements, `std::experimental::generator<>` is limited to use with resumable functions.

This is type erasure at the finest level of granularity. Every value produced by the generator requires an indirect function call. By embedding fine-grained type erasure into the language, we are denied the opportunity to develop library abstractions that amortise the cost of runtime polymorphism. For example, given direct access to the implementation type `__foo`, we could write an efficient polymorphic wrapper, let's call it `batching_generator<int>`, where the indirect function call cost is incurred once for every N elements.

4.4 Scalability of layered abstractions

Abstractions are a key tool in managing software complexity. Consider, for example, a typical asynchronous networking scenario that involves a message protocol with a fixed-length header and variable-length body. Rather than implement the protocol logic in a single, complex resumable function, we wish to structure the tree of operations as follows:

⇒ Read message
- ⇒ Read header
  - ⇒ Read N bytes
    - ⇒ Read data off socket
- ⇒ Read body
  - ⇒ Read N bytes
    - ⇒ Read data off socket

Each operation represents a level of abstraction, and has its own set of post-conditions that must be satisfied before its continuation can be invoked. For example, the "Read N bytes" resumable function exists to manage the problem of partial reads (where a socket returns fewer bytes than requested), and the function cannot complete until the requested number of bytes is read or an error occurs.

Rather than minimising the abstraction penalty, a consequence of N4402 resumable functions' type erasure is that they exhibit poor scalability as we add layers of abstraction. Each layer requires its own "stack frame" which must be allocated. Although there is some capacity to customise this allocation, it requires the cooperation of the callee. And, as we saw in the previous section, as each resumable function returns up the "stack" it incurs the cost of an indirect function call to resume the parent coroutine.

Under the suspend-down model, each layer of abstraction is a simple function call, and includes the opportunity for inlining. The runtime cost of adding additional abstraction layers is minimised, and in many cases zero.

4.5 When scheduling logic is embedded in the language

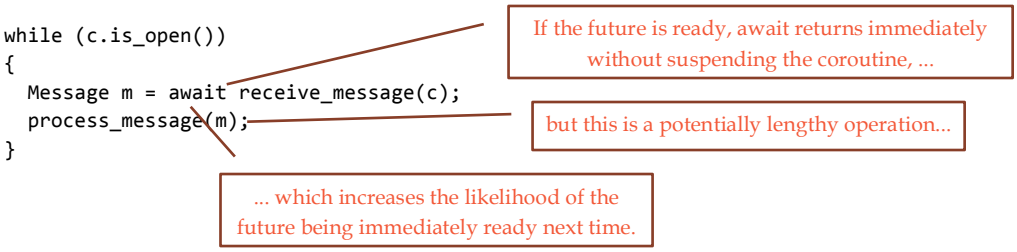
Let us assume we have an asynchronous function to receive a message for a single connection:

```
future<Message> receive_message(Connection& c);
```

Now, let's write a function to receive messages and process them:

```
future<void> process_messages(Connection& c)
{
  while (c.is_open())
  {
    Message m = await receive_message(c);
    process_message(m);
  }
}
```

The problem here is that we have no way of knowing if `receive_message` will return a future that is immediately ready. If there is an already buffered or queued message, it may well do so. This means that our message processing loop:



may block the calling thread indefinitely.

This design, where `await` prefers to avoid coroutine suspension if it can, is a **bad** default. It leads to unfairness, jitter, and starvation. It makes programs susceptible to denial-of-service by badly behaved or hostile peers.

Can we defend against this? According to N4499, this behaviour is controlled by the `await_ready` function. If `await_ready` returns `false`, the coroutine is suspended; if it returns true then the coroutine continues without suspending.

We need `await_ready` to always return false, but as we can't customise `await_ready` for `future<>`, we will create an adapter template.

```
template <class T>
struct always_suspend_t {
  T t;
};

template <class T>
auto always_suspend(T t) {
  return always_suspend_t<T>{ std::move(t) };
}

template <class T>
bool await_ready(always_suspend_t<T>&) noexcept
{
  return false;
}
```

We must also forward the `await_suspend` and `await_resume` functions. This turns out to be decidedly non-trivial, since to follow the language rules for `await` as described in N4499, we must forward to either member or non-member functions. Here is an attempt for `await_suspend`:

```
template <class T>
void await_suspend_fwd(T& t, std::experimental::coroutine_handle<> h, std::true_type)
{
  t.await_suspend(h);
}
```

```
template <class T>
void await_suspend_fwd(T& t, std::experimental::coroutine_handle<> h, std::false_type)
{
  await_suspend(t, h);
}

template <class T>
void await_suspend(always_suspend_t<T>& a, std::experimental::coroutine_handle<> h) noexcept
{
  await_suspend_fwd(a.t, h,
    std::integral_constant<bool,
      sizeof(await_suspend_memfn_helper<T>(0)) != 1>());
}
```

Lots of SFINAE helper stuff not shown here

Now that we have our adapter, we can update our `await` expression to use it:

```
Message m = await always_suspend(receive_message(c));
```

It turns out we are not done yet. Even if `await_ready` returns false, `await_suspend` could still result in the coroutine being resumed immediately. The `await_suspend` function for a `future<>` will be implemented in terms of `future<>::then()`, and `future<>::then()` can call its continuation on any thread. Presumably that includes the current thread.

Unfortunately, the author did not get as far as implementing a solution for this. One possible solution would involve a thread local variable and passing an intermediate function object to `future<>::then()`. This is left as an exercise for the reader.

So, assuming we solve all of the above issues, are we done now? Alas, no. We still have to worry ourselves about the implementation of `receive_message`. It may also be implemented as a resumable function, in which case we need to make sure its `await`s also use the `always_suspend` adapter. It turns out that, to write safe resumable functions, our `always_suspend` adapter is as viral as the `await` keyword.

All this risk and complexity is a consequence of embedding what is essentially a scheduling decision – to continue a coroutine immediately – inside a language keyword. Resumable functions should be used to simplify the composition of asynchronous operations, and scheduling should be treated as a cross cutting concern. Potentially dangerous design decisions like this should not be hard-wired into the language, but instead left to the library – where they can be fixed or, perhaps, worked around by a user providing their own implementation.

5 A way forward

The problems presented above are purely associated with the *up-and-out* model of resumable functions. Stackful coroutines, which suspend *down*, do not suffer from these issues.

N4232 asserts that *up-and-out* is in the very nature of stackless coroutines, but is that necessarily so? Might we be able to have stackless coroutines that, syntactically, use the *down* model of suspension?

6 Evaluation modes in C++

C++14 currently has two modes of evaluation:

1. Evaluation during program execution, on the abstract machine.
2. Evaluation during translation, but following the rules of the abstract machine.

The latter mode is, of course, that used by *constant expressions*.

More concretely, when we write:

```
auto x = expression ;
```

the compiler has the opportunity to evaluate *expression* using either mode. If it is able, then as an optimisation it will evaluate it during translation. Otherwise, the value assigned to `x` will be determined during program execution.

However, when we write:

```
constexpr auto x = expression ;
```

the compiler has no such freedom. By specifying `constexpr` we have constrained the compiler to the mode where evaluation occurs during translation. If it is unable to do so, the program is ill-formed and the compiler issues a diagnostic. Put another way, the `constexpr` keyword in the above statement "activates" the evaluation mode for constant expression.

7 The resumable expressions approach

7.1 Underlying design principle

As of today, a C++ compiler can take the statement:

```
constexpr auto x = expression ;
```

and, during translation, transform *expression* into a constant that is named `x`. With C++14 we also have relaxed `constexpr`, and *expression* may include calls to functions that include loops, branches, and so on.

We can think of `constexpr` as a special case of a more general form. Given a suitable identifying keyword, it is also feasible for a compiler to read:

```
keyword type x = expression ;
```

and subsequently transform *expression* into some other form during translation.

7.2 Keyword `resumable` and the representation of a resumable object

Resumable expressions are identified by a new keyword, `resumable`:

```
resumable auto r = expression ;
```

This declares a new variable `r`, where `r` is a resumable object. The type is `auto` as the compiler generates a corresponding class definition for `r`, of an unspecified type *`R`*, which looks like this:

```
class R
{
public:
  typedef decltype( expression ) result_type;
  bool ready() const noexcept;
  void resume() noexcept(noexcept( expression ));
  result_type result();
  R(const R&) = delete;
  R& operator=(const R&) = delete;
} r{unspecified};
```

The key features of this class are described below.

```
typedef decltype( expression ) result_type;
```

Identifies the type of the expression. Expressions of type void are allowed.

```
bool ready() const noexcept;
```

Returns true only when evaluation of the expression has completed and the result of the evaluation is available.

```
void resume() noexcept(noexcept( expression ));
```

Requires that `ready() == false`. Resumes evaluation of the expression so that it executes until it reaches the next suspension point. Throws only if the expression throws, i.e. any exception thrown by the expression is allowed to propagate to the caller.

```
result_type result();
```

Requires that `ready() == true`. Returns the result of the evaluation. May be called at most once.

```
R(const R&) = delete;
R& operator=(const R&) = delete;
```

The "stack" of the resumable function is stored as a data member of the class *R*. A resumable function may take the address of its local variables and store them across suspension points. Copying and moving is inhibited to prevent this local variable aliasing from causing issues.

```
r{unspecified};
```

Any variables used in the expression are captured by reference.

7.3 Resumable objects as data members

Resumable objects are non-copyable and non-moveable, but we still want to be able to use them as data members and allow them to reside within heap-allocated objects. To enable this, the `resumable` keyword can be applied to non-static data members, provided the data member has an initialisation expression.

For example, the following template allows a zero-argument resumable function to be captured and allocated on the heap:

```
template <class F>
struct resumable_wrapper
{
  explicit resumable_wrapper(F f) : f_(f) {}
  F f_;
  resumable auto r_ = f_();
};

template <class F>
resumable_wrapper<F>* alloc_resumable_wrapper(F f)
{
  return new resumable_wrapper<F>(std::move(f));
}
```

7.4 Keyword `resumable` applied to functions

The `resumable` keyword may be specified on function definitions:

```
resumable void foo()
{
  ...
}
```

The `resumable` keyword may also be specified on a function declaration. If a function is declared or defined `resumable` then all declarations and definitions must specify the `resumable` keyword.

Like `constexpr` or `inline` functions, functions marked as `resumable` may be defined in more than one translation unit.

The `resumable` keyword may not be used on virtual functions.

7.5 The `break resumable` statement

A statement of the form

```
break resumable;
```

marks a suspension point in a resumable function. However, a resumable function is not required to contain a `break resumable` statement. Instead, it may call another function that contains the statement. For example:

```
resumable void foo()
{
  break resumable;
}

resumable void bar()
{
  foo();
}
```

A `break resumable` statement may occur only when evaluating a resumable expression, otherwise the program is ill-formed. This means that a function containing this statement cannot be called in non-resumable contexts.

7.5.1 Resumable and non-resumable functions

All functions, including template specializations and template instantiations, can be classified as either resumable or non-resumable.

Any function declared with the `resumable` keyword is resumable.

For inline functions and functions defined within a class definition, the `resumable` keyword may be omitted. The function is resumable if it contains a `break resumable` statement or a call to another resumable function.

For lambdas, the `resumable` keyword is not used. A lambda's function call operator is resumable if it contains a `break resumable` statement or a call to another resumable function.

For function templates and member functions of a class template, the `resumable` keyword may be omitted. When template instantiation occurs, a determination can be made as to whether a function is resumable or non-resumable. It is resumable if it contains a `break resumable` statement or a call to another resumable function. This permits templates to make calls to other potentially resumable functions, dependent on a template parameter.

For a function template specialization to be resumable, it must either be inline or be declared with the `resumable` keyword. Extern templates cannot be resumable.

For polymorphic lambdas, the function call operator is considered a function template and treated as described above.

All other functions are non-resumable.

7.6 Restrictions on resumable functions

When a resumable function is used in a resumable expression, the definition of the function must appear before the end of the translation unit.

You cannot take the address of a resumable function.

Resumable functions cannot be virtual.

Resumable functions cannot be recursive.

7.7 New evaluation mode

Let us introduce a new, third mode of evaluation:

3. Transformation to a resumable object during translation. Evaluation of the resumable object occurs during program execution.

This is the evaluation mode used for resumable expressions. When the compiler encounters a statement of the form:

```
resumable auto x = expression ;
```

it activates the resumable expression evaluation mode in order to transform the expression.

7.8 Evaluation mode rules

Whether a particular function can be called depends on the active evaluation mode. For example, non-`constexpr` functions cannot be called when in the evaluation mode used for constant expressions. The corresponding rules for resumable functions are shown in the table below.

Active evaluation mode	**Call to a resumable function**	**Call to a non-resumable function**
Program execution	Ill-formed.	OK. The function definition is also evaluated in program execution mode.
Constant expression	Ill-formed.	OK if the function is marked `constexpr`, otherwise ill-formed. The function definition is evaluated in constant expression mode.
Resumable expression	OK. The function definition is evaluated in resumable expression mode.	OK. The function definition is evaluated in program execution mode. (This means that such a function can never be part of a suspended resumable object's call stack.)

7.9 Nested resumable expressions

A resumable function is allowed to contain a resumable expression. For example:

```
resumable void inner(int n)
{
  for (int i = 1;;)
  {
    std::cout << i << std::endl;
    if (++i > n) break;
    break resumable;
  }
}
```

```
resumable void outer(int n)
{
  for (int i = 1; i <= n; ++i)
  {
    resumable auto r = inner(i);
    while (!r.ready())
    {
      std::cout << "inner resume ... ";
      r.resume();
      break resumable;
    }
  }
}

int main()
{
  resumable auto r = outer(5);
  while (!r.ready())
  {
    std::cout << "outer resume ... ";
    r.resume();
  }
}
```

However, a resumable object's `resume()` member function is always non-resumable. This allows us to nest resumable expressions without creating any confusion as to which resumable object a `break resumable` statement applies to. Each resumable expression is a self-contained object and the compiler does not need to consider the nested resumable expression as part of the outer resumable expression's "call stack".

7.10 Restrictions on the standard library

Standard library functions and template specializations are resumable if, and only if, they are explicitly specified to be so using the `resumable` keyword. Standard library templates may be instantiated as either resumable or non-resumable functions unless the `resumable` keyword is specified.

8 Implementation strategy

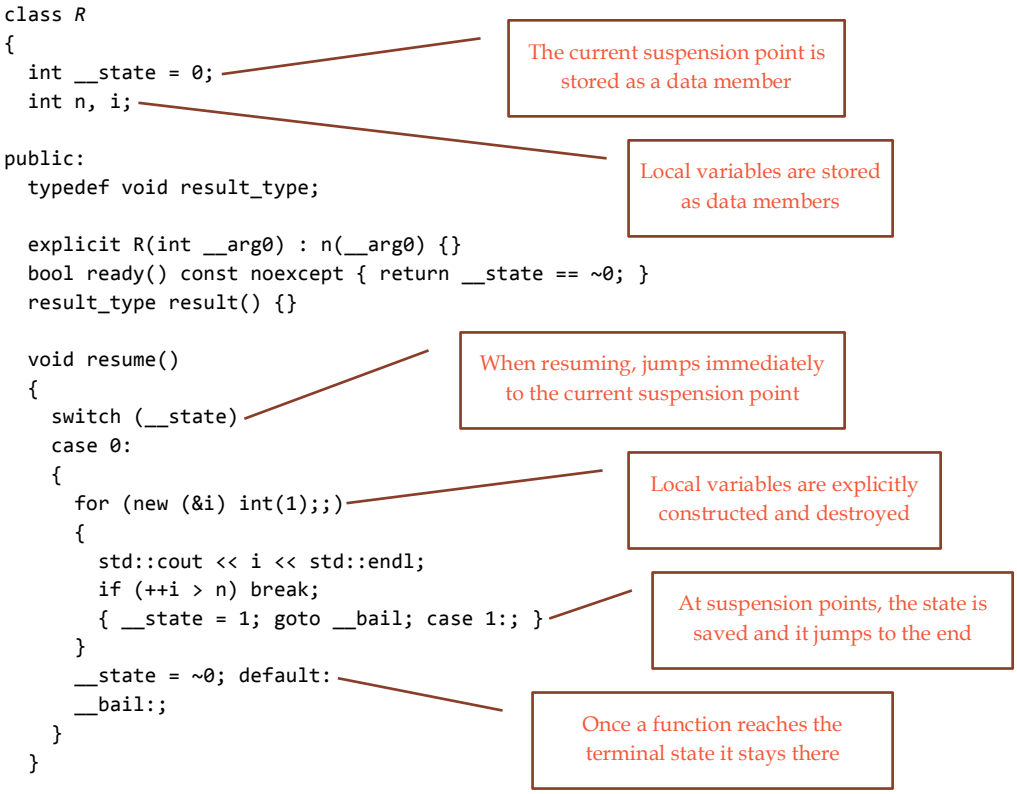
Given a resumable function:

```
resumable void print_1_to(int n)
{
  for (int i = 1;;)
  {
    std::cout << i << std::endl;
    if (++i > n) break;
    break resumable;
  }
}
```

that is used in a resumable expression:

```
resumable auto r = print_1_to(5);
```

the compiler may generate the following code to represent the resumable object `r`:



9 Alternative syntax

9.1 Creation of a resumable object

A criticism of resumable expressions, as proposed above, is that it can be unclear which variables are being captured when a resumable object is created. An alternative would be to adopt a lambda approach for creating a resumable object, as proposed in N4244 Resumable Lambdas.

For example, rather than:

```
resumable auto r = print_1_to(n);
```

where n is implicitly captured by reference, we may write:

```
auto&& r = [n] resumable { print_1_to(n); };
```

The body of the resumable lambda would be evaluated in resumable expression mode.

9.2 Yielding values

Another issue with resumable expressions is that they complicate the development of otherwise trivial generators. As the only suspension primitive provided is:

```
break resumable;
```

some kind of library abstraction is a virtual necessity in order to yield a value, such as that illustrated in section 12.2 below.

Rather than `break resumable`, an alternative approach could be to use a `co_yield` keyword, similar to that used in N4402 and N4244. When used alone:

```
co_yield;
```

the effects are identical to `break resumable`. However, `co_yield` could also be used with an operand:

```
co_yield X;
```

If combined with the lambda creation syntax above, simple resumable objects are identical to N4244 resumable lambdas in form and function:

```
auto&& r = [] resumable -> int {
  for (int i = 0; i < 5; ++i)
    co_yield i;
};
```

Unlike N4244 resumable lambdas, as resumable expressions use the suspend-down model, we now have the ability to yield values from nested function calls:

```
resumable void yield_ints(int n)
{
  for (int i; i < n; ++i)
    co_yield i;
}

auto&& r1 = [] resumable -> int { yield_ints(5); };
```

or:

```
auto&& r2 = [] resumable -> int {
  std::vector<int> v = { 1, 2, 3, 4, 5 };
  std::for_each(v.begin(), v.end(), [](int i){ co_yield i * 2; });
};
```

As resumable expressions suspend down, for non-trivial uses of resumable expressions the impact of these syntax changes may be limited to library developers. These changes have little to no effect on application code using these facilities. For example, the application code shown in sections 12.5 to 12.9 is unaffected by these changes.

10 Further work

10.1 Allowing copyability

By allowing copyability of resumable objects, we enable interesting use cases such as undo stacks. Although this behaviour comes with risk associated with aliasing of local variables, an explicit opt in may be feasible.

10.2 Overloading on resumable

It may be worth allowing multiple overloads of a function that differ only in the presence of the `resumable` keyword. The selected overload is determined based on whether the current evaluation mode is for a resumable expression or not.

11 Prototype

A prototype of the resumable expressions language extension may be found at:

https://github.com/chriskohlhoff/resumable-expressions

This prototype emulates the runtime behaviour of resumable expressions using the Boost.Coroutine library and suitably defined pre-processor macros. As a consequence there are minor differences in syntax:

Proposed syntax	**Emulated syntax**

`break resumable;`	`break_resumable;`
`resumable auto r = expression;`	`resumable_expression(r, expression);`

Furthermore, the prototype does not enforce the compile-time rule that `break resumable` statements appear only in resumable expressions.

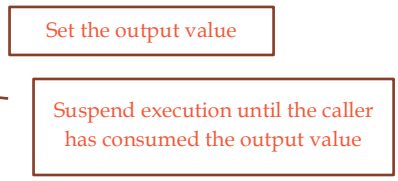
12 Examples

Please note that resumable expressions are a pure language feature. The examples below are for illustrative purposes only, and the library abstractions presented are not part of the proposal.

12.1 A simple generator

The following example is a simple generator, implemented using the resumable expression primitives directly.

```
resumable void fib(int n, int& out)
{
  int a = 0;
  int b = 1;
  while (n-- > 0)
  {
    out = a;
    break resumable;
    auto next = a + b;
    a = b;
    b = next;
  }
}
```



It is callable like this:

```
int out;
resumable auto r = fib(10, out);
while (!r.ready())
{
  r.resume();
  if (!r.ready())
    std::cout << out << std::endl;
}
```

However, for typical usage patterns, like generators, we do not want to have to use these low level primitives. Fortunately, resumable expressions allow us to build library abstractions on top. For generators, there are two parts to this:

- An abstraction for the act of yielding a value.
- An abstraction for a generator object, to allow the generated values to be consumed.

The following two sections illustrate how we might implement this library abstraction.

12.2 Implementing “yield” in a library

To implement generators, we first want an abstraction for yielding a value. The following example shows one possible approach:

```
template <class T>
struct yielder
{
  T& out;
  void operator()(T t)
  {
    out = t;
    break resumable;
  }
};
```

No `resumable` keyword required on inline functions – the use of `break resumable` automatically makes it a resumable function

This is then used to implement our `fib()` function:

```
resumable void fib(int n, yielder<int> yield)
{
  int a = 0;
  int b = 1;
  while (n-- > 0)
  {
    yield(a);
    auto next = a + b;
    a = b;
    b = next;
  }
}
```

Execution of the resumable function is suspended inside the function call

12.3 Type-erased generators and separate compilation

The definition of a resumable function must appear before the end of any translation unit in which it is used. What if we want our `fib()` function to be separately compiled? One solution is to introduce a type-erased generator wrapper. This wrapper will present a generator as an object from which we can conveniently consume the generated values.

```
// Declaration in .h file:
generator<int> fib(int n);

// Definition in .cpp file:
generator<int> fib(int n)
{
  return {
    [=](auto yield) mutable
    {
      int a = 0;
      int b = 1;
      while (n-- > 0)
      {
        yield(a);
        auto next = a + b;
        a = b;
        b = next;
      }
    }
  };
}
```

A type-erased wrapper for generators

A function object, similar to the previous `yielder<T>` example, that contains the suspension point

Lambdas are automatically considered resumable if they contain a call to a resumable function

What follows is one possible implementation for this type-erased wrapper:

```
// An exception to be thrown when the generator
// completes and has no further values
class stop_generation : std::exception {};
```

```
// The base class for all generator implementations
template <class T>
struct generator_impl_base
{
  virtual ~generator_impl_base() {}
  virtual T next() = 0;
};

// A concrete generator implementation
template <class T, class G>
struct generator_impl : generator_impl_base<T>
{
  generator_impl(G g)
    : generator_(g) {}

  virtual T next()
  {
    r_.resume();
    if (r_.ready())
      throw stop_generation();
    return value_;
  }

  G generator_;
  T value_;
  resumable auto r_ = generator_(
      [v = &value_](T next) {
        *v = next;
        break resumable;
      });
};
```

Resume until the user-supplied generator type yields the next value

The resumable expression has been fully evaluated, meaning the generator has no more values

A resumable object may be a data member, provided it has a resumable expression as a member initializer

The user-supplied generator object is called with a lambda that abstracts away the yield operation, similar to the previous `yielder<T>` example

```
// The type-erased generator wrapper
template <class T>
class generator
{
public:
  template <class G>
  generator(G g)
    : impl_(std::make_shared<generator_impl<T, G>>(std::move(g))) {}

  T next() { return impl_->next(); }

private:
  std::shared_ptr<generator_impl_base<T>> impl_;
};
```

To add support for allocators, we simply need to provide an additional constructor that accepts an allocator:

```
// The type-erased generator wrapper
template <class T>
class generator
{
public:
  // ...
  template <class G, class Allocator>
  generator(std::allocator_arg_t, const Allocator& a, G g)
    : impl_(std::allocate_shared<generator_impl<T, G>>(a, std::move(g))) {}
  // ...
};
```

Using the allocator is simply a matter of calling the appropriate `generator<T>` constructor. The allocator does not have to be part of the `fib()` function's parameter list.

```
// Definition in .cpp file:
generator<int> fib(int n)
{
  return {
    std::allocator_arg, my_allocator_obj,
    [=](auto yield) mutable
    {
      // ...
    }
  };
}
```

Now that our generator wrapper is complete, we can use the separately compiled `fib()` as follows:

```
int main()
{
  try
  {
    auto gen = fib(10);
    for (;;)
      std::cout << gen.next() << std::endl;
  }
  catch (stop_generation&)
  {
  }
}
```

12.4 Generic generators

So far our generator examples have been resumable functions. However, we want to be able to write our algorithms so that they are reusable in both normal and resumable contexts. To do this, we simply write our algorithm as a template, and make the usage of `break resumable` dependent on a template parameter.

```
template <class OutputIterator>
void fib(int n, OutputIterator out)
{
  int a = 0;
  int b = 1;
  while (n-- > 0)
  {
    *out++ = a;
    auto next = a + b;
    a = b;
    b = next;
  }
}
```

No `resumable` keyword required on function templates

Whether or not it is resumable depends on calls that involve the template parameter

This generic generator can be used in normal, non-resumable code:

```
std::vector<int> v;
fib(10, std::back_inserter(v));
for (int i: v)
  std::cout << i << std::endl;
```

Or, with a suitable iterator:

```
template <class T>
struct resumable_iterator
{
  T& out;
  suspending_output_iterator& operator*() { return *this; }
  suspending_output_iterator& operator=(T t)
  {
    out = t;
    break resumable;
    return *this;
  }
  // Other members as required by OutputIterator requirements
};
```

we can use our generic iterator as part of a resumable expression:

```
int out;
resumable auto r = fib(10, resumable_iterator<int>{out});
for (;;)
{
  r.resume();
  if (r.ready()) break;
  std::cout << out << std::endl;
}
```

12.5 Implementing "await" in a library

Let us assume we want to use an asynchronous operation that signals its completion via a future. For example, we may have a function that "sleeps" for a specified length of time, declared as:

```
future<void> sleep_for(std::chrono::milliseconds ms);
```

This particular incarnation of `future<>` allows us to attach a continuation using `then()`:

```
sleep_for(std::chrono::millseconds(500)).then([]{ /* ... */ });
```

This approach quickly becomes unwieldy as our sequence of continuations evolves to more than a simple, linear chain. Therefore, we would like to be able to use this, and other future-based operations, within a resumable function. This would allow us apply the control flow constructs with which we are all familiar – `if`, `while`, `for`, function calls, and so on – to the problem of composing asynchronous operations.

To do this, we will define a function called `await()` that suspends the calling resumable function until a future is ready. The `await()` function would be used as in the following example:

```
resumable void print_1_to(int n)
{
  for (int i = 1;;)
  {
    std::cout << i << std::endl;
    if (++i > n) break;
    await(sleep_for(std::chrono::milliseconds(500)));
  }
}

int main()
{
  spawn([]{ print_1_to(10); }).get();
}
```

The resumable function is suspended here until the future becomes ready

What follows is one possible implementation for the `await()` function. We will begin by implementing a base class, `waiter`, for the representation of a running resumable function.

```
class waiter :
  public std::enable_shared_from_this<waiter>
{
public:
  virtual ~waiter() {}

  void run()
  {
    struct state_saver
    {
      waiter* prev = active_waiter_;
      ~state_saver() { active_waiter_ = prev; }
    } saver;
    active_waiter_ = this;

    std::lock_guard<std::mutex> lock(mutex_);
    nested_resumption_ = false;
    do_run();
  }

  resumable void suspend()
  {
    assert(active_waiter_ == this);
    if (!nested_resumption_)
    {
      active_waiter_ = nullptr;
      break_resumable;
    }
  }

  void resume()
  {
    if (active_waiter_ == this)
      nested_resumption_ = true;
    else
      run();
  }

  static waiter* active()
  {
    return active_waiter_;
  }

private:
  virtual void do_run() = 0;

  std::mutex mutex_;
  bool nested_resumption_ = false;
  static __thread waiter* active_waiter_;
};
```

This function runs the resumable function until it is suspended by waiting on a future

A thread-local variable keeps track of the active resumable function

A mutex ensures that the resumable function can run from only one thread at a time

This member is used to detect when an `await()` call completes immediately

The derived class implements this virtual function to run the concrete resumable function

This function suspends the resumable function until it is woken by a call to `resume()`

It may be called only by the currently active resumable function

As long as there has been no nested call to `resume()`, then the resumable function suspends itself

This function resumes a resumable function when the future becomes ready

If the current thread is already running the resumable function, indicate that a nested resumption has occurred

Otherwise, call `run()` to resume running the resumable function from this thread

Next we have a class template, derived from `waiter`, as the concrete representation of a resumable function.

```
template <class F>
class waiter_impl : public waiter
{
public:
  explicit waiter_impl(F f) :
    f_(std::move(f)) {}

private:
  virtual void do_run()
  {
    while (active() == this && !r_.ready())
      r_.resume();
  }

  F f_;
  resumable auto r_ = f_();
};
```

This function is called from `waiter::run`

It keeps the resumable expression executing until it waits on a future or until the expression is fully evaluated

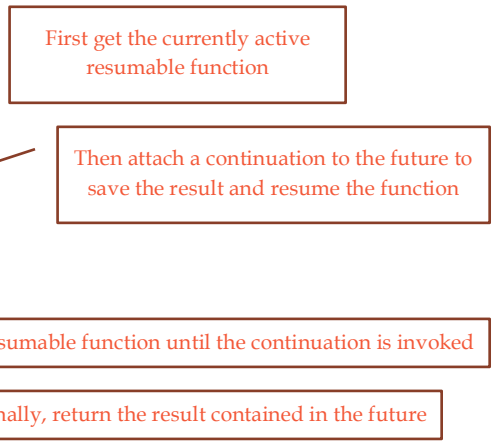
A resumable object that evaluates the function object f_ as a resumable expression

Next we implement a `spawn()` function to create a new `waiter` for a given resumable function, start it, and return a future to allow us to wait for it to complete.

```
template <class F>
void launch_waiter(F f)
{
  std::make_shared<waiter_impl<F>>(std::move(f))->run();
}

template <class Resumable>
auto spawn(Resumable r,
    typename std::enable_if<
      std::is_same<
        typename std::result_of<Resumable()>::type,
        void
      >::value
    >::type* = 0)
{
  promise<typename std::result_of<Resumable()>::type> p;
  auto f = p.get_future();

  launch_waiter(
      [r = std::move(r), p = std::move(p)]() mutable
      {
        try
        {
          r();
          p.set_value();
        }
        catch (...)
        {
          p.set_exception(std::current_exception());
        }
      });

  return f;
}
```

The implementation of `await()` is straightforward.

```
template <class T>
resumable T await(future<T> f)
{
  waiter* this_waiter = waiter::active();
  assert(this_waiter != nullptr);

  future<T> result;
  f.then([w = this_waiter->shared_from_this(),
          &result](auto f)
      {
        result = std::move(f);
        w->resume();
      });

  this_waiter->suspend();
  return result.get();
}
```



12.6 Separate compilation of asynchronous operations

As noted above, the definition of a resumable function must appear before the end of any translation unit in which it is used. Should we wish to use separate compilation for our resumable function, we can trivially do so by using the `spawn()` function defined above.

```
// Declaration in .h file:
future<void> print_1_to(int n);

// Definition in .cpp file:
future<void> print_1_to(int n)
{
  return spawn([n]
      {
        for (int i = 1;;)
        {
          std::cout << i << std::endl;
          if (++i > n) break;
          await(sleep_for(std::chrono::milliseconds(500)));
        }
      });
}
```

We can then compose this operation using our `await()` function.

```
resumable void print_numbers()
{
  await(print_1_to(10));
  await(print_1_to(5));
}
```

12.7 Efficient composition of asynchronous operations

When separate compilation is not required, we can achieve more efficient composition by simply calling other resumable functions directly, just as you would any other function.

```
resumable void print_1_to(int n)
{
  for (int i = 1;;)
  {
    std::cout << i << std::endl;
    if (++i > n) break;
    await(sleep_for(std::chrono::milliseconds(500)));
  }
}
```

```
resumable void print_numbers()
{
  print_1_to(10);
  print_1_to(5);
}
```

Using the "normal" function call idiom for composition brings several advantages:

- it limits the viral proliferation of `await()` throughout our source code;
- we do not have to worry about forgetting to call `await()`; and
- it is as cheap as an inlined function call.

Furthermore, since function templates may be used in either resumable or non-resumable contexts, we can easily reuse generic algorithms.

```
resumable void print_numbers()
{
  std::vector<int> v = { 1, 2, 3, 4, 5 };
  std::for_each(v.begin(), v.end(), [](int n){ print_1_to(n); });
}
```

12.8 Integrating asynchronous operations that use completion handlers

Asynchronous operations using Boost.Asio's *completion token* mechanism[1] may be seamlessly integrated by building on the same infrastructure as `await()`.

```
resumable void echo(tcp::socket socket)
{
  for (;;)
  {
    char data[1024];
    size_t n = socket.async_read_some(buffer(data), use_await);
    async_write(socket, buffer(data, n), use_await);
  }
}
```

When we pass the completion token `use_await`, the initiating function suspends the current resumable function until the operation completes

Once the operation completes, the initiating function returns the result of the operation

To implement this we begin by defining the completion token type.

```
constexpr struct use_await_t
{
  constexpr use_await_t() {}
} use_await;
```

Our completion token has a corresponding completion handler type:

```
template <class... Args>
struct await_handler;
```

which we associate with our completion token by specializing the `handler_type` trait. This trait generates a completion handler type given a completion token and a call signature.

```
template <class R, class... Args>
struct handler_type<use_await_t, R(Args...)>
{
  typedef await_handler<Args...> type;
};
```

For brevity, we will show only the `await_handler<>` partial specialization that implements a completion handler that has an `error_code` as its first argument.

[1] See N4045 *Library Foundations for Asynchronous Operations, Revision 2*

```
template <class... Args>
struct await_handler<std::error_code, Args...>
{
  await_handler(use_await_t) {}

  void operator()(const std::error_code& ec, Args... args)
  {
    if (ec)
      *this->exception_ = std::make_exception_ptr(std::system_error(ec));
    else
      this->result_->reset(std::make_tuple(std::forward<Args>(args)...));

    this->waiter_->resume();
  }

  typedef std::tuple<std::decay_t<Args>::type...> tuple_type;
  std::shared_ptr<waiter> waiter_;
  boost::optional<tuple_type>* result_ = nullptr;
  std::exception_ptr* exception_ = nullptr;
};
```

The function call operator is invoked when the asynchronous operation completes

Any error is captured as an exception

Otherwise, the result is packaged in a tuple

Then we can resume running the resumable function

Next, we require some helper functions to unpack the tuple that contains the result. This result will be returned from the initiating function.

```
template <class... T>
inline std::tuple<T...> get_await_result(std::tuple<T...>& t)
{
  return std::move(t);
}

template <class T>
inline T get_await_result(std::tuple<T>& t)
{
  return std::move(std::get<0>(t));
}

inline void get_await_result(std::tuple<>& t)
{
}
```

If the tuple contains two or more elements we use it as-is

If the tuple contains just one element we use that

For empty tuples we return void

We finish by specializing the `async_result` trait, which is used to associate the completion handler with the initiating function's return type and value.

```
template <class... Args>
class async_result<await_handler<Args...>>
{
public:
  typedef typename await_handler<Args...>::tuple_type tuple_type;
  typedef decltype(get_await_result(std::declval<tuple_type&>())) type;

  explicit async_result(await_handler<Args...>& handler)
  {
    assert(waiter::active() != nullptr);
    handler.waiter_ = waiter::active()->shared_from_this();
    handler.result_ = &result_;
    handler.exception_ = &exception_;
  }
```

This constructor is called just before the asynchronous operation starts

Store the current resumable function (`waiter`), and space for the result, into the completion handler

```
  resumable type get()
  {
    waiter::active()->suspend();

    if (exception_)
      std::rethrow_exception(exception_);
    return get_await_result(result_.get());
  }

private:
  boost::optional<tuple_type> result_;
  std::exception_ptr exception_;
};
```

This function is called just before the initiating function returns

Suspend the resumable function until the completion handler is invoked

Return the result of the asynchronous operation

12.9 Generic asynchronous operations

Ideally, we want to be able to compose our asynchronous operations as generic algorithms that work equally well with resumable expressions, stackful coroutines, and normal threads. The solution is the same as we saw above for generators: we write our algorithm as a template, and make the usage of `break resumable` dependent on a template parameter.

We can achieve this by defining new *synchronous completion token* type requirements, which are a refinement of P0112's completion token requirements. These new requirements behave as illustrated in the code below.

```
template <class SyncCompletionToken>
void echo(tcp::socket& socket, SyncCompletionToken token)
{
  for (;;)
  {
    char data[1024];
    size_t n = socket.async_read_some(buffer(data), token);
    async_write(socket, buffer(data, n), token);
  }
}
```

A synchronous completion token

When the token is passed to an asynchronous operation, the initiating function suspends until the operation is complete ...

... and then returns the result

The `use_await` completion token we implemented above is one implementation of the synchronous completion token requirements.

```
echo(my_socket, use_await);
```

Boost.Asio's stackful coroutine support provides another implementation.

```
[](yield_context yield) {
  // ...
  echo(my_socket, yield);
}
```

Finally, N4045 section 7.5, *Synchronous execution contexts,* shows a possible synchronous completion token implementation that blocks the current thread.

```
echo(my_socket, block);
```

13 Acknowledgements

The author would like to thank Jamie Allsop, Botond Ballo, Thorsten Ottosen, Geoffrey Romer, Nevin Liber, and Nate Wilson for their feedback on this or previous revisions of this proposal.