Document number: P1430R1

Date: 2019-02-08
Reply To:

Mihail Mihaylov (mmihailov@vmware.com)

Vassil Vassilev (v.g.vassilev@gmail.com)
Audience: WG21 Evolution Working Group

First-class symmetric coroutines in C++

Changelog

Changes from R0

- Fixed the asynchronous coroutine example in section 4.4.5, thanks to feedback from Gor Nishanov.
- Added a section discussing generators and ranges.
- Added a complete generator example in the appendix.
- Updated the comparison with Regular Expressions.
- Added a sentence to the introduction clarifying that this is work in progress.
- Added text to section 4.1 clarifying that the definition syntax has issues and will change in the future.
- Added a paragraph to section 4.4.3 to clarify the semantics of `get_caller()`.
- Made minor fixes to typos in several examples.
- Added Asen Krastev and Lewis Baker to the acknowledgements section.

1. Introduction

The Coroutines TS [1], Core Coroutines [2], and Resumable Expressions [3] proposals represent different tradeoffs and design decisions in the stackless coroutines design space. While they have both minor and major differences, they all have one thing in common - they all provide asymmetric transfer of control (however the Coroutines TS and Core Coroutines simulate symmetric control through a trampoline).

Here we present the current progress of our work on a design for first-class, type-safe symmetric coroutines for C++ based on a general model of coroutines as a generalization of regular functions.

Similarly to the Core Coroutines proposal, our design represents coroutines as classes. Programmers can create instances of the coroutine class and can control where to allocate them. Unlike the Core Coroutines proposal, the coroutine definition specifies a base class for the coroutine class. The base class can be either a built-in coroutine class or a user-defined class. This provides many opportunities for customizations.

Unlike the other proposals, the transfer of control in our proposal is purely symmetric. In addition, the transfer of control operation also transfers data. This allows coroutines in our design to be used in place of the wrapper and awaitable helper objects that the Coroutines TS and Core Coroutines design require.

We compare the other designs to our design and discuss the specific constraints and tradeoffs represented by each design.

2. Coroutines as a generalization of functions

While a function is being evaluated it keeps some temporary runtime state associated with the specific invocation of this function, consisting of the function's local variables and control flow related information. This runtime state is called an activation record. When multiple invocations of the same function are in progress there is a separate activation record for each of these invocations.

When the function invokes another function, the caller preserves its activation record, suspends its execution and transfers control to the callee, potentially along with some data. The callee uses its own activation record to maintain its runtime state. When the callee completes, its activation record is destroyed, and the caller resumes its execution. The caller may also receive some data when it resumes execution.

The suspension and resumption capabilities of regular functions are limited in a number of ways:

- The activation record of a function invocation, either suspended or running, cannot be referenced explicitly - the program cannot create and destroy activation records, or obtain their addresses, or pass them as arguments, etc. Activation records are manipulated implicitly when functions call each other.
- Control flow is strictly nested - a function invocation can only transfer control “up” to its caller or “down” to a new function invocation that it instantiates.
- A function invocation cannot transfer control to another function invocation more than once - even when a function calls another function multiple times, each call creates a new activation record of the target function.
- The transfer of control to the callee cannot be delayed - the program cannot create an activation record and use it at a later time.
- When the callee transfers control back to its caller, it can no longer be resumed, because its activation record is destroyed as soon as it returns control to its caller.

Coroutines generalize on regular functions by eliminating all of these limitations. A coroutine is an object which represents the activation record of a function invocation. It has its own lifetime at runtime and a set of applicable operations. The programmer can:

- Create the coroutine object explicitly and then reference this object explicitly.
- Allocate the coroutine object in dynamic storage to extend its lifetime.
- Transfer control to any suspended coroutine - not just the caller of the current coroutine or a new coroutine created by the current one.
- Transfer control to the coroutine repeatedly at any time during the coroutine’s lifetime or not at all.

This gives programmers greater control over the execution flow of the program - they are not limited to the strictly nested calls of regular functions. The call stack is replaced by the combination of a set of suspended coroutines and a single active coroutine. At any time the currently active coroutine can resume any of the suspended coroutines.

Coroutines also generalize the meaning of the function call operation and the meaning of function types. With regular functions, the function type `X(Y)` means a function which takes an argument of type `Y` and produces a result of type `X`. But a coroutine may never return to its caller and the value that the caller expects may be produced by an arbitrary other coroutine.

So when a coroutine `foo()` invokes another coroutine `bar()` defined as “`X bar(Y)`” what this actually means is that:

- `bar()` is a suspended coroutine which expects to get a value of type `Y` when it is resumed.
- `foo()` is transferring control to `bar()` along with a value of type `Y`.
- `foo()` suspends itself, and expects to get a value of type `X` when it gets resumed later.

3. Functions as a special case of coroutines

Let’s picture a programming language where all functions are full-featured coroutines. Even with coroutines being one of the primary control flow mechanisms, we can still expect that the majority of coroutine invocations in user code will be strictly nested like regular functions. It will then make sense for the compiler to detect this very common use case and perform specialized optimizations. In particular the strictly nested lifetimes allow the activation records of these coroutines to be allocated on a dedicated stack instead of on the heap. Furthermore, there is no need to provide explicit references to these coroutine objects, because the compiler can determine the target of each transfer of control at compile time.

The compiler can detect when a set of coroutine invocations is strictly nested and lay them out using a standard call stack. In other words, we can view regular functions as coroutines optimized for a very common special case.

4. A general purpose design for first-class coroutines for C++

Here we present our design proposal for coroutines support in C++. There are several questions that each coroutines design for C++ should answer:

1. How to define coroutines.
2. How to instantiate them and manage their lifetime.
3. How to reference them.
4. How to transfer control between them.
5. How to exchange data between them.
6. How to customize their behavior.

On one hand, these design decisions will affect the ease of use of coroutines in different use cases, and will affect optimization opportunities on the other.

We believe that a design which makes coroutines first-class objects will compose best with the other parts of the language. It will have the lowest risk of usability problems in unexpected use cases. Such design is future proof because it is more likely to work well even for use cases that we are unable to foresee now and because it is closer to the already built mental models of the C++ programmers.

4.1. Defining coroutines

A coroutine definition needs to define both a coroutine body and a factory that creates coroutine instances with that body. Most first-class objects in C++ are defined as classes or types and their instances are created and destroyed using constructors and destructors, so it makes sense for the coroutine definition to define a class with the coroutine body as a method of the class.

One possible approach would be to use the class definition syntax with a specially named method. We prefer not to use this approach, because it is unintuitive and more verbose.

Our proposal is to use the function definition syntax, but have it define a class when the function is a coroutine. In the current text of the proposal, the compiler will interpret a function definition as a coroutine definition when the return value is of the built-in `coroutine` class or a class that inherits it. The compiler-generated class will have the name of the function used to define it and will derive from the return type. It will have a constructor with the same parameters as the coroutine definition and will bind them to private variables with the same names. The template argument of the base `coroutine` class will determine the types of the values passed to the coroutine and yielded by the coroutine.

For example, the coroutine definition:

```
coroutine<T(U)> foo(A a, B b) {
   // ....
}
```

will generate the class:

```
class foo : public coroutine<T(U)> {
   A a;
   B b;
   union {
      // local variables for all suspension points
   };

   T __body(U) {
      // ....
   }

public:
   // The foo class passes the body to coroutine<T(U)> with an explicit
   // call instead of passing it to the constructor to make it easier
   // to derive from the coroutine class.
   foo(A a, B b) : a(a), b(b) { coroutine<T(U)>::__set_body(&__body); }
   ~foo() {}
};
```

`T(U)` specifies how the generated coroutine is invoked after its creation. The arguments a and b are passed to the constructor and injected into the generated class as private members. They are visible via unqualified lookup to the coroutine body which is transformed into a private method of the generated class (named `__body` in the example).

The purpose of the current syntax is to convey the semantics of our proposal. We have received feedback proposing better options. We will provide a better definition syntax in a later revision of this document.

The base `coroutine` class provides the common functionality shared by all coroutines and the compiler-generated class provides the definition of the coroutine state and the coroutine body.

One benefit of this design is that when coroutines are regular classes they can be used in every way a class can. They can be instantiated, they can be templates, they can be passed by reference and they can participate in overload resolution, template specialization and template instantiation.

Another benefit is that the special operations available in the body of a coroutine like `yield()` are just methods of the base coroutine class. This reduces their visibility and reduces the risk of clashes with user identifiers. Furthermore, the user can provide additional functionality to the coroutine body by deriving a new class from the base coroutine class.

Last but not least, the option to derive user-defined classes from the `coroutine` class makes it possible to define specialized coroutine types, which extend or modify the public coroutine interface and provide additional functions to the body of the coroutine.

The drawback is that the layout of the coroutine frame, or at least its size, becomes public and the compiler cannot optimize it in the general case. It may be possible to find ways to address this that will not sacrifice the benefits of representing coroutines as named classes.

Alternatives

We have discussed alternatives for the definition syntax and semantics and may change them in the future based on feedback and usage experience. Richard Smith made two suggestions:

1. Use the same definition syntax, but instead of a class, generate a factory function that returns instances of an anonymous type that derives from the return type in the original definition. This will provide the same generality as our current proposal, but might be more intuitive and will address a number of challenges.
2. Introduce a new syntax to indicate more explicitly that the definition defines a coroutine: `auto foo(A a, B b) : coroutine<T(U)> { }`. This can define either a class or a factory function, so it's orthogonal to the first suggestion.

Challenges

The challenges to exposing the coroutine state as a first class object are due to the fact that the size of the activation record depends on the optimizer, so it cannot be determined correctly in the frontend. As a consequence:

- The coroutine class layout can be different in different translation units, leading to ABI problems.
- On the other hand fixing the coroutine class layout in the frontend will prevent many optimization opportunities.

We believe that in many cases the compiler will be able to determine that it’s safe to defer fixing the coroutine class layout to the optimizer and that programmers will be able to use this optimization to achieve performance close to the performance of type-erased coroutines. We are still exploring the possibilities.

Further work

During our experiments with the design we noticed the recurring of the pattern of storing a coroutine inside an object (another coroutine or a regular class) to use it as a method of the class. However, as coroutines are separate objects, such a coroutine needs to keep a pointer to the owning object, which is inefficient.

We expect that this pattern will be common, so we intend to look for a suitable syntax to specify that a coroutine will be embedded into another object. Then the compiler cam emit appropriate code which uses hardcoded offsets to locate the members of the owning object.

4.2. Coroutine instantiation and lifetime

With coroutines represented as first-class named classes both instantiation and lifetime management become easy and natural - we just use the ordinary construction syntax. Coroutine objects can be allocated on the stack or on the heap and can be aggregated into other classes. Coroutines on the stack and aggregated as members of other objects benefit fully from RAII. When programmers need to extend the lifetime of a coroutine they just allocate it on the heap with `new`, `make_shared` or `make_unique`.

When a coroutines is created it is suspended at an implicit suspension point immediately before the start of the user-defined coroutine body. It can be resumed by calling operator() on the instance:

```
coroutine<int(void)> counter(int start) {
   int i = start;

   while (true)
      yield(i++);
}

int main() {
   counter c1(0); // Automatic variable
   auto c2 = new counter(10); // Raw pointer
   auto c3 = make_unique<counter>(20); // unique_ptr

   for (int i = 0; i < 5; ++i)
      cout << c1() << ", " << (*c2)() << ", " << (*c3)() << ", " << endl;

   delete c2;
}
```

The output of this example program is:

```
0, 10, 20
1, 11, 21
2, 12, 22
3, 13, 23
4, 14, 24
```

The only special thing about coroutine objects is that in the general case they are non-copyable and non-moveable, because the coroutine frame is a part of the object, and some local variables can be references or pointers to other local variables.

4.3. Referencing coroutines

Instances of first-class coroutines can be referenced like any other object. They can be passed by reference or pointer, they can be cast to a base class, etc.

```
void foo(counter& c1, coroutine<int(void)>* c2) {
   for (int i1 = c1(), i2 = (*c2)();
        !(c1.done() || c2->done() || i1 > 10);
        i1 = c1(), i2 = (c2)()) {
      // ...
   }
}
```

```
int main() {
   counter c1(0);
   counter c2(10);

   foo(c1, &c2);
}
```

4.4. Transferring control and data between coroutines

The operation that transfers control and data to a regular function is the function call operator. It is also the operation that transfers control and data to a classic functor. So it seems natural to use the function call operator with coroutines as well. On the other hand, it can be argued that a dedicated word that makes it explicit that the target of the call is a coroutine will improve code readability. Currently our proposal uses the function call operator, but it can use a distinct word just as easily.

4.4.1 Resume continuations

A suspended coroutine is represented by a “resume continuation” - a function-like object which resumes the coroutine when invoked. The suspended coroutine receives the value passed to the resume continuation and continues execution from the last point where it was suspended. It also receives implicitly the resume continuation of the calling coroutine or regular function.

The transfer of control is completely symmetric - the only way for a coroutine to suspend itself is to invoke a resume continuation which will resume another coroutine. Therefore, resume continuation invocations mark the suspension points in the coroutine. The compiler is required to: transform the coroutine body in a way which will ensure that every suspension point is a tail call position by preserving in the activation record all local variables that live across the suspension point; and to generate code for the resume continuation invocation which performs a tail call.

The `coroutine` object itself is callable too, because it contains a resume continuation. Invoking the `coroutine` object resumes its execution. When a coroutine gets resumed via an invocation of its `coroutine` object, it can call `get_caller()` to obtain a resume continuation which will return control to the caller. The `yield()` method and the `return` statement are syntactic sugar for invoking the continuation returned by `get_caller()` and provide an asymmetric interface on top of the symmetric core of the design. For more information on `get_caller()` see section 4.4.3.

As described in section 2, the type of the resume continuation depends on the type of the value that the callee expects and on the type of the value that the callee can expect if it passes control back to the original coroutine.

Suppose we have two coroutines foo() and bar():

```
coroutine<X(Y)> bar() {
   // ....
   get_caller()(X());
}

coroutine<T(U)> foo() {
   // get_initial_value() returns the first value passed to foo
   U u = get_initial_value();
   // ....
   X x = bar()(Y());
   return T();
}

int main() {
   foo f;
   T t = f(U());
}
```

The instance of `foo()` in `main()`, f has an operator "`T operator()(U)`" that `main()` calls. This operator calls the initial resume continuation stored in `f`; `main()` suspends itself and passes the control and a value of type `U` to `f`. When `main()` is resumed it will receive a value of type `T`.

Then `f` creates an instance of another coroutine `bar()` and transfers control to it. It passes to it a value of type `Y` and a resume continuation.

At this point, the invocation of `bar()` can no longer call `f` as a function of type `T(U)` to resume it, because `f` is not expecting a `U`, it's expecting an `X`. Therefore, the resume continuation, that `f` passes implicitly to the instance of `bar()`, should be represented as a function object which takes an `X`.

As for the return type of the resume continuation of `f`, when `bar()` transfers control back to `f`, it expects to get an `Y` when it is resumed again. This means that the type of the resume continuation is `Y(X)`.

As this example demonstrates, the type of the resume continuation of the caller is determined by exchanging the positions of the argument type and return type of the callee. This ensures that the argument type of the continuation represents the type that the caller expects to receive on resumption. An intuitive way to think about this is as a pair of typed channels between the two coroutines - what's the input channel on one side is the output channel on the other side and vice versa.

When the coroutine is first instantiated, the argument type of its initial continuation is the one given in the coroutine definition. The following example illustrates this further. The type of the continuation that each suspension point passes implicitly to the target coroutine is specified in the comments:

```
coroutine<int(void)> counter() {
   int i = 0;
   while (true)
      yield(i++); // Continuation type: int(void)
}

coroutine<string(const char*)> bar() {
   // get_initial_value() returns the first value passed to bar
   const char* s = get_initial_value();

   counter c;
   while (c() < 9) // Continuation type: void(int)
      s = yield(string(s)); // Continuation type: string(const char*)

   return string(s); // No continuation
}

int main() {
   bar b;
   while (!b.done())
      string s = b("a"); // Continuation type: const char*(string)
}
```

Alternatives

The current design allows only single-argument coroutines, because passing multiple arguments to a coroutine will need to be reflected by yield() returning multiple values in the coroutine body. We could support multi-argument coroutines if we define that some resume continuations will return a tuple or a tuple-like type.

Further work

The current design always stores a resume continuation in the coroutine object. The motivation is that the code that creates a coroutine object needs to obtain both a resume continuation and an object representing the activation record. On the other hand, for cases when the coroutine will only be resumed through its object once in the beginning, this seems like a waste. We plan to explore possibilities to separate the continuation of the initial suspend point from the activation record object.

Further work

The semantics of the resume continuation is very close to that of `unique_function`. It seems that it should really be move only. We also plan to explore possibilities to also use it as a `unique_ptr` analogue to manage lifetime for heap-allocated coroutines.

4.4.2. Invalidation of resume continuations

The resume continuation represents a suspension point in the coroutine. Invoking it invalidates it because the coroutine may never be suspended at this place again - it may complete or it may suspend itself at a different suspension point (where it may even expect a different type of value). And it even may not be the original coroutine that the caller invoked that returns control to it. When the caller is resumed, it cannot use the same continuation object to resume safely the same coroutine. Instead, the resume continuation object is updated to reflect the actual coroutine which transferred control back to the caller:

```
using cont = resume_continuation<void(int)>;

coroutine<int(cont)> foo() {
   // get_initial_value() returns the first value passed to foo
   cont c = get_initial_value();
   for (int i = 0; ;++i)
      c(i);
}

coroutine<int(void)> bar() {
   foo f;
   int i = f(get_caller());
   return i;
}

coroutune<int(void)> baz() {
   bar br;
   int sum = br();
   for (int i = 1; i < 10; ++i)
      sum += br();
   return sum;
}
```

```
int main() {
   baz bz;
   return bz();
}
```

In this example:

1. `main()` creates an instance of `baz()` `bz` and invokes it.
2. `bz` creates an instance of `bar` `br` and transfers control to it. The resume continuation of `bz` is of type `void(int)`.
3. `br` creates `f` - an instance of `foo()` - and transfers control to it. It also explicitly passes `bz`'s resume continuation to `f`.
4. Then `f` uses this continuation to yield an `int` value. This has three effects:
    a. Suspends `f` and resumes `bz`.
    b. Passes an `int` value to `bz`.
    c. Stores the resume continuation of `f` in `br`.
5. On the next iteration of the loop, when `bz` invokes `br`, this actually passes control to `f`, bypassing `br`. It also modifies `c`, so that the next call to `c` in the loop in `f` will pass control back to the loop in `bz`.
6. Control will go back and forth between the loop in `bz` and the loop in `f` and `br` will never receive control again.

Using a resume continuation before its last user has been resumed is undefined behavior.

4.4.3 Initial value and caller

When a resume continuation is invoked, the value passed to it is passed to the target coroutine as the result of the yield expression, and the resume continuation of the source coroutine overwrites the resume continuation used by the yield expression in the target coroutine.

But coroutines are created in a suspended state. This means that when the coroutine is resumed for the first time there is no yield statement in it to receive the initial value and the resume continuation of its first caller. To receive this first value, the coroutine needs to call the `get_initial_value()` method. To receive the resume continuation of the first caller, it needs to call `get_caller()`.

In this example:

```
template<typename V> coroutine<V(V)> sum() {
   V sum = 0;
   V x = get_initial_value();
   while (true) {
      sum += x;
      x = get_caller()(sum);
   }
}

coroutine<int(int)> root_of_sum_of_squares() {
   // Implicit initial suspension point
   sum<double> adder;
   int x = get_initial_value();
   while (true) {
      double sum = adder(double(x) * double(x)); // Suspension point 1
      // yield(...) is syntactic sugar for get_caller()(...)
      x = yield(int(sqrt(sum))); // Suspension point 2
   }
}
```

`root_of_sums_of_squares` expects a `double` at suspension point 1, but expects an `int` at suspension point 2 and at the initial suspension point.

Note that `sum` calls `get_caller()` multiple times. The first time `get_caller()` returns the resume continuation of the first caller of the coroutine, and after that it returns the resume continuation of the coroutine which resumed it after it last used `get_caller()` to transfer control.

The return value of `get_caller()` is a reference to a regular resume continuation and follows the same rules outlined in section 4.4.1 and section 4.4.2 as any other resume continuation. Therefore its return value will not change unless the coroutine invokes it. Invoking the resume continuation returned by `get_caller()` always transfers control to the last suspended coroutine or the regular function which resumed the current coroutine through the coroutine object itself.

4.4.4. Root coroutine

As regular functions in C++ are not full-featured coroutines, the first coroutine executed by a program always needs to be invoked by a regular function, for example `main()`. The activation record of this regular function is at the top of the system call stack. As described in 4.4.1., the transfer of control from one coroutine to another is a tail call that doesn't grow the stack; if the coroutine consumes some stack while it's running it must free it before suspending itself. Therefore, if the first coroutine invokes another coroutine, the second coroutine will be executed with the same call stack as the first one. The same holds for coroutines invoked by the second coroutine and coroutines invoked by these other coroutines and so on.

Suspending a coroutine must always transfer control to some other code, so when all coroutines are suspended, the control must ultimately return to the original regular function. We can view this regular function as a special kind of coroutine as described in section 3. This kind of coroutines allow only nested control flow; their activation records are allocated on the system stack and only the one on the top of the stack can be resumed. We will refer to this special coroutine as the current “root coroutine”.

If a coroutine calls a regular function, and the callee calls another coroutine, the new coroutine is invoked in the context of the nested regular function which becomes the new root. In other words, the root coroutine is always the regular function invocation at the top of the call stack.

There are usage scenarios when a coroutine which wasn't invoked directly by the root coroutine needs to transfer control to it, bypassing the other coroutines in the current chain of invocations. One such use case is asynchronous I/O and other asynchronous workflows. There are also use cases that require non-local return, including parsers.

However, a coroutine that was not invoked directly by the root coroutine has no information about the type of value that the root coroutine expects. Therefore, coroutines that transfer control to the root coroutine, or call other coroutines that may transfer control to the root, need to specify the type of the root explicitly. This allows the compiler to verify and enforce that the transfer of control is type-safe. We specify the type of the resume continuation of the current root coroutine as an optional second function-type template argument of the coroutine type and the resume continuation type. Coroutines can only invoke resume continuations with the same root coroutine type as their own.

Any coroutine that has specified its root coroutine type can call the `get_root_coroutine()` method to get a resume continuation which returns control to the current root. This transfer of control follows the standard rules for all coroutines and updates the resume continuation that the root coroutine used last. When the root coroutine uses this resume continuation again, it passes control back to the last suspended coroutine.

For example this code:

```
coroutine<double(double), int(int)> foo() {
   double d = get_initial_value();
   cout << "foo() started with: " << d << endl;
   int i = get_root_coroutine()(int(d));
   cout << "foo() resumed with: " << i << endl;
   return double(i)
}
```

```
coroutine<int(int), int(int)> bar() {
   int i = get_initial_value();
   cout << "bar() started with: " << i << endl;
   double d = foo()(double(i));
   cout << "bar() resumed with: " << d << endl;
   return int(d);
}

int main() {
   cout << std::fixed << std::setprecision(1);
   bar b;
   int i = b(1);
   i = b(2);
   cout << "bar() returned: " << i <<endl;
}
```

will print:

```
bar() started with: 1
foo() started with: 1.0
foo() resumed with: 2
bar() resumed with 2.0
bar() returned 2
```

Alternatives

As defined currently, the root coroutine type parameter of the `coroutine` template class might be counterintuitive, especially because it follows the opposite convention to the first one. The first template parameter specifies how the coroutine will be invoked by direct users of the `coroutine` object, and the type returned by `get_caller()` is deduced by exchanging the argument type and return type.

In contrast, we have defined the second template parameter to specify the type returned by `get_root_coroutine()`. It will be more consistent and probably more intuitive to define this parameter to be the type of the resume continuation that the root coroutine will use to resume the current coroutine. Then the type returned by `get_root_coroutine()` will be the reverse continuation type similar to `get_caller()`.

4.4.5. Detaching a coroutine

However, in the fire-and-forget (asynchronous) use case we don't want the coroutine to be resumed until it's ready. For this use case we provide the `detach()` operation. It invokes the resume continuation in the regular way but doesn't pass the continuation of the current coroutine to the target.

After a coroutine has detached itself, it can no longer be resumed. So it has the same semantics as a return statement and indeed the return statement in a coroutine body is syntactic sugar for detaching to the caller of the coroutine.

The following example demonstrates the usage of `detach()` for asynchronous I/O:

```
using read_resumer = resume_continuation<void(tuple<error_code, size_t>),
                                         void(void)>;

coroutine<void(void), void(void)>
resume_reader(read_resumer cont, error_code ec, size_t count)>) {
   detach(cont, {ec, count});
}

// This could evolve into a generic awaiter-like coroutine
coroutine<tuple<error_code, size_t>(), void(void)>
suspend_reader(Stream& s, Buffer& buffer) {
   read_resumer cont = get_caller();

   async_read(s, buffer, [cont](error_code ec, size_t count) {
      resume_reader(cont, ec, count)();
   });

   detach(get_root_coroutine());
}

coroutine<void(void), void(void)> count_bytes(Stream& s) {
   size_t total_count = 0;
   Buffer buffer(1024);

   while (true) {
      auto [ec, count] = suspend_reader(s, buffer)();
      if (ec) {
         return;
      total_count += count;
   }
}
```

```
int main() {
   Stream s(/* ... */);

   count_bytes bytesCounter(s);
   bytesCounter();

   /* ... */
}
```

In this example, `main()` creates the `count_bytes` coroutine `bytesCounter` and starts it. `bytesCounter` creates a temporary `suspend_reader` coroutine and passes control to it. The `suspend_reader` coroutine binds the resume continuation of `bytesCounter` to a lambda and passes the lambda as a callback to the `async_read` operation. Then it detaches itself and returns control to `main()`. `main()` can no longer resume `bytesCounter` which will now be driven asynchronously by callbacks. When the `async_read` operation completes, it executes the lambda, which creates and invokes a temporary `resume_reader` coroutine, which resumes `bytesCounter`. The `resume_reader` coroutine is needed to preserve the type-safety of `get_root_corouitne()`.

This is also thread-safe, because:

- By the time the resume continuation of `bytesCounter` is bound to the callback, it is already suspended, therefore suspension and resumption cannot race.
- The body of the `async_reader` coroutine doesn't access any of its members after the call to `async_read()`. The root continuation is on the top of the stack, so `get_root_coroutine()` doesn't need to access the state of the current coroutine. And `detach()` is an intrinsic which just passes control to the target suspension point without accessing the state of the current coroutine.

Another related use case is cooperative multitasking. We can implement a cooperative switch operation as a coroutine which pushes the resume continuation of its caller in the end of the dispatcher queue and then calls `detach(get_root_coroutine())` to give control to the main dispatcher loop. The dispatcher loop then pops the next resume continuation from the front of the queue and executes it.

4.5. Customization points

With coroutines represented as classes, the natural way to customize their allocation is by overriding the `new` and `delete` operators.

As for customizing the coroutine-related aspects of their behaviour, the two natural ways are to either overriding methods of the class, or to specialize a traits template. Some behavior customizations could also be achieved by wrapping the coroutine into adapter classes.

The disadvantage of customizing the coroutine behaviour by overriding methods is that they need to be overridden in a base class, because the actual class of the coroutine is automatically generated by the compiler. In contrast, the traits approach allows to customize the behavior even of the compiler-generated class.

On the other hand, inheritance allows us to write mixins that provide additional functionality in the coroutine body. Customizing the coroutine aspects of the class with overrides in the base class would be consistent with that.

An important point regarding customization points defined as methods of a base class is that they don't need to be virtual - they would be called by the generated class, which will also be final, so static dispatch will work.

We still haven't decided on the set of customization points. Some customization methods that we have discussed are:

`void on_suspend()`

Called after the coroutine is suspended. Should return a coroutine to allow it to perform a tail call to another coroutine. Will need to be a template to handle suspend points of different types.

`void on_resume()`

Called after the coroutine has been resumed. Will need to be a template to handle suspend points of different types.

`void on_init()`

Called immediately after the coroutine is created in the context of the code which created it. The most obvious use of this suspension point is to create coroutines that start their execution on creation. Like `on_suspend()`, it would need to return a temporary coroutine to be able to execute a tail call.

`void on_detach()`

Called when a coroutine completes by calling `detach()` or executing a `return` statement. Such a customization point can be used to automatically destroy the coroutine when it completes execution.

`void get_yield_target() / void get_return_target()`

Called when a coroutine calls `yield()` or executes a return statement respectively to determine the resume continuation to invoke. It's not clear if they would be of any use.

4.6. The base coroutine class and the resume continuation class

The compiler will provide two built-in class templates - `resume_continuation` and `coroutine`.

The `resume_continuation` is a function-like type which represents a suspension point and provides an operation that transfers control and data to the suspended coroutine. It specifies both the suspension point and the activation record of the coroutine. The `resume_continuation` class is a template parametrized by one or two function types:

```
template<class...> class resume_continuation;

template<class R> class resume_continuation<R(void)>;

template<class R, class A> class resume_continuation<R(A)>;

template<class R, class RR> class resume_continuation<R(void), RR(void)>;

template<class R, class RR, class RA>
class resume_continuation<R(void), RR(RA)>;

template<class R, class A, class RR> class resume_continuation<R(A), RR()>;

template<class R, class A, class RR, class RA>
class resume_continuation<R(A), RR(RA)>;
```

It provides three methods:

`operator()`

This is actually an intrinsic. The return type and argument type of the `operator()` are determined by the first function type template argument.

Calls to this operator are replaced with code which performs the following steps:

1. Preserve the current state of the coroutine in the activation record.
2. Place the argument and the current continuation where the target coroutine will look for them.
3. Invalidate this continuation and jump to the target coroutine.
4. Restore the coroutine state.
5. Copy the resume continuation of the source coroutine into this continuation.

Steps 1 and 2 are the pre-suspend sequence, similar to the epilogue of a regular function. Steps 4 and 5 are the post-resume sequence, similar to the prologue of a regular function. Steps 1-3 are executed in the source coroutine (the currently active coroutine) when it yields control. Steps 4-5 are executed in the target coroutines when it receives control.

This sequence behaves as if the resume continuation is a functor that has bound the target coroutine and performs a call to a method on the target coroutine but with two distinctions from a regular this-call:

- Instead of a call, it performs a jump.
- Instead of saving the return address on the call stack, it passes the resume continuation of the current coroutine as an additional hidden argument.

The variants of `resume_continuation` that have the second function type template argument can only be called from coroutines that derive from the coroutine class with the same type as a second template argument.

```
bool done() const
```

Returns `true` if the coroutine which this continuation corresponds to has executed a `return` statement. The continuation is valid, but can no longer be invoked.

```
bool valid() const
```

Returns `true` if the continuation is valid. If it returns `false`, the other two methods cannot be invoked.

The `coroutine` class is the base class for all coroutine instances. Like the `resume_continuation` template it can be instantiated as either `coroutine<R(A)>` or `coroutine<R(A), RR(RA)>`.

A `coroutine` object stores a `resume_continuation` with the same template parameters as itself and provides the same interface as the continuation: `operator()`, `done()` and `valid()`. These methods call the corresponding methods of the stored continuation. They are always inlined.

Further work

We are considering adding a method or cast operator that extracts the `resume_continuation` stored in the `coroutine` object, as well as method that, given a `resume_continuation` provides a base `coroutine` pointer.

The `coroutine` class also provides four protected method that can be used in the coroutine body:

`resume_continuation<A(R)>& get_caller()`

Initially this method returns the continuation of the coroutine or regular function which first resumed the function through the coroutine object. If the coroutine invokes the caller continuation, control returns to the first resumer of the coroutine. Then, when the coroutine is resumed again (necessarily through the coroutine object), the continuation is updated.

`A get_initial_value()`

Only available if the coroutine accepts an argument. Returns the value that was passed to the coroutine the first time it was resumed.

`resume_continuation<RR(RA)>& get_root_coroutine()`

Only available for coroutines defined with the second function type template argument. Returns a resume continuation that passes control to root coroutine (the regular function at the top of the call stack).

```
template<class R, class A>
void detach [[ noreturn ]] (resume_continuation<R(A)>, A)
```

This is an intrinsic too. Resumes the suspended coroutine corresponding to the resume continuation but doesn't pass the current continuation to it. After a coroutine uses this method it can no longer be resumed.

The intended use of this method is for asynchronous I/O workflows and other “fire and forget” coroutines.

`A yield(R)`

This is also an intrinsic. It serves as syntactic sugar for the expression `A get_caller()(R)`.

`return R`

The return statement is equivalent to a jump to an implicit suspension point right after the body of the coroutine. It destroys all local variables, marks the coroutine as complete and yields to the resume continuation returned by `get_caller()`. As a result it has the natural behavior expected of it.

Further work

We are considering adding a `tail_call()` intrinsic method. It would be almost identical to `detach()`.The difference is that `detach()` doesn't send the continuation of the current coroutine to the target coroutine, whereas `tail_call()` will send a continuation specified by the programmer.

Further work

We are considering adding an assignment operator that only takes a temporary coroutine of the exact same type (this shouldn't be a problem if the generated coroutine classes are final). This operator would have the meaning of resetting the coroutine - i.e. it would restart it with new creation arguments.

5. Generators and ranges

5.1. Generators

One prominent use case for coroutines is the implementation of generators. At first glance, they are a natural fit. The `done()` method can be used as an indicator for the end of the sequence like in:

```
coroutine<int(void)> generator() { /* ... */ }

for (generator g; !g.done(); ) {
    cout << g() << endl;
}
```

But the coroutine needs to be invoked at least once, which will necessarily produce a value. Therefore this approach cannot produce an empty sequence. This demonstrates that the function call interface of coroutines is not suitable for generators, because it needs to express both the produced value and the end of the generated sequence.There are various approaches that can address this shortcoming:

- The coroutine can throw an exception to signal the end of iteration.
- It can designate a value within the domain of generated values as a sentinel value and return it at the end of the sequence.

- It can expand the type of the generated values with a sentinel value. For example, it can return an `optional<T>` or `expected<T>`.
- It can return the value in an output argument and use the return value only to signal the end of the sequence.
- Similar to the previous approach, the coroutine can yield void and use additional storage as a side channel to store the last generated value and done() can signal the end of the sequence.

The appendix contains an example that implements a generic base class for generators implemented as coroutines following the last approach. Some of the other approaches are presented in the following examples:

```
const int sentinel = -1;
coroutine<int(void)> sentinel_generator() { /* ... */ }

sentinel_generator sg;
for (int val = sg(); val != sentinel; val = sg()) {
   cout << val << endl;
}
```

```
coroutine<optional<int(void)> optional_generator() { /* ... */ }

optional_generator og;
for (optional<int> val = og(); val.has_value(); val = og()) {
   cout << val.value() << endl;
}
```

```
coroutine<bool(/* out */ int&)> output_argument_generator() { /* ... */ }

output_argument_generator oag;
int val;
while (oag(val)) {
   cout << val << endl;
}
```

5.2. Ranges

It's easy to implement a range on top of any of the generator interfaces presented above and in the appendix. Unfortunately, the range is not a coroutine itself and using the generator through a range will involve a call to a regular function, which in turn calls the generator coroutine.

This poses a relatively minor problem in the synchronous case - using the range will add activation records on the system call stack, which may blow the stack in case of deep recursion, but the result will be no worse than using a non-coroutine generator.

The problem is much worse when using an asynchronous coroutine-based generator from another coroutine. In essence, an asynchronous coroutine is a coroutine which arranges to be resumed at a later time and returns control to the root coroutine (the regular function at the top of the call stack). When a coroutine uses directly a generator which is implemented as an asynchronous coroutine, the generator passes control to the caller's root and the caller remains suspended, as expected. But when the generator is used through a range, the range introduces a new root between the generator coroutine and its caller coroutine. When the generator suspends itself, it passes control to this new root, which has no other choice but to return to its caller coroutine. The end result is that the caller coroutine is resumed before the value that it's expecting is ready, leading to unpredictable incorrect behavior.

This is a problem with all stackless coroutine designs. It's also not limited to ranges, but extends to all regular functions that call asynchronous coroutines. It can only be solved by making these utilities coroutine-aware, or introducing some form of "suspend-down" support.

6. Comparison to other coroutine designs

6.1. Coroutines TS

There are three main differences between the Coroutines TS design and ours:

- The coroutine state is not a first-class object.
- Transferring control doesn't transfer data.
- The call chain is not preserved.

6.1.1. The coroutine state is not a first-class object

The TS makes the coroutine state an internal implementation detail, hidden behind a type-erased handle. This allow the compiler to perform a lot of optimizations and only generate the coroutine frame layout after that, leading to a very efficient representation.

The disadvantage of this approach is that programs can no longer use coroutines like objects in many ways:

- Coroutine objects cannot be allocated explicitly in automatic storage.
- The coroutine state cannot be aggregated into other objects.
- The actual coroutine type cannot be used for type-based dispatch, only the wrapper type can.

- Consequently, programmers cannot use the actual type of the coroutine to customize the behavior of the coroutine.

However, the design provides the ability to wrap coroutines in helper objects which are used in place of an explicit coroutine type. This still doesn't provide automatic storage for the coroutine frame and the TS relies on the compiler to optimize away the allocations.

6.1.2. Transferring control doesn't transfer data

The coroutine_handle::resume() method is the actual control transfer mechanism in the TS. As it accepts and returns void, coroutines in the TS have gained the ability to be suspended and resumed, but they have lost the ability to use the standard function call mechanisms to exchange data.

Instead, the transfer of control operator co_await is a trampoline which uses user-defined helper objects (promises and awaitables) to transport values. While this is a natural fit for the asynchronous use cases, it is not as good a fit for the synchronous use case. Furthermore, even in the the asynchronous use case, the disconnect between transfer of control and transfer of control makes this approach unsafe and error-prone.

6.1.3. The call chain is not preserved

TS coroutines don't preserve their call chain - resuming a coroutine through the original coroutine handle resumes the actual coroutine and not the last suspended coroutine in its chain. This is both surprising and unsafe - a coroutine can be resumed by code which was not intended to resume it, resulting in unexpected behavior.

While user code can use a suitable wrapper to preserve the call chain this solution seems far from ideal.

6.1.4. Trade-offs comparison

The trade-offs that the Coroutines TS makes are:

- Optimize the coroutine frame layout at the cost of making coroutines "second-class" and potentially allocating memory for scoped coroutines.
- Less general operation of control transfer.

To compensate for these trade-offs, the TS associates the coroutine with a first-class object, and relies on heap-elision optimisations in the compiler.

The trade-off that our design makes is to make coroutines first-class and powerful enough to not require helper objects at the price of fixing the size of the coroutine at an early stage of the compilation, and potential ABI problems.

We believe that the compiler will be able to detect situations when it can postpone the generation of the coroutine frame layout, and we are looking actively for an approach that would address the ABI issue.

6.2. Core Coroutines

There are three main difference between the Core Coroutines design and ours:

- Coroutines always start executing on creation.
- Coroutines always return a wrapper.
- The transfer of control is asymmetric.

We expect that with our design programmers will use helper coroutines in place of many of the uses of the unwrap operator.

However, at this point our understanding of the Core Coroutines design is insufficient for a comparison of the trade-offs and usage patterns. We plan to add a more informed comparison in the next revision of this document.

6.3. Resumable expressions

Resumable expressions present a design that’s fundamentally different from the Coroutines TS and Core Coroutines, as well as from our design. The key differences from our design are:

- Regular functions are transformed silently into resumable expressions, which allows them to become a part of the coroutine’s control flow.
- The transfer of control is asymmetric. Unlike the Coroutines TS and Core Coroutines, they don’t use a trampoline to simulate a call chain.
- Code is transformed in a way which combines multiple resumable functions into one to simulate suspension across several invocations.
- As a consequence, resumable expressions don’t support recursive invocations of coroutines.
- Resumable expressions can only be allocated in automatic storage. To allocate them on the heap, they need to be wrapped explicitly in another object.

We believe that resumable expressions address the very important issue of using coroutines together with regular functions. This allows a lot of pre-existing non-coroutine code to work with coroutines without modifications - most notably ranges and the standard library. However they do so at the expense of providing a very limited transfer of control operation.

We believe that our design provides a more powerful and convenient transfer of control operation. At the same time we want to explore different approaches to integrate coroutines with regular functions including the ideas in the Resumable Expressions design. However, in our opinion, adopting our design will not preclude the pursuit of solutions to the integration problem and we can work on it at a later stage.

6.4. Stackful coroutines

There are different interpretations of the term “stackful coroutines”. In some sense even our design could be called “stackful” because it preserves the invocation chain by default. What we mean by stackful coroutines in this document are designs which provide symmetric transfer of control between complete standard call stacks as described in “fiber_handle - fibers without scheduler” [4].

The stackful approach is very different from all the other approaches in that it provides a single handle for a whole chain of invocations. This leads to two important differences from our design:

- The suspend operation preserves the whole execution context of the coroutine, including the activation records of non-coroutine functions.
- Only the last activation in the invocations chain can be resumed.

The main advantage of the stackful approach is that it can be introduced into pre-existing code bases with minimal modifications to the pre-existing code.

The two main drawbacks are that:

- The symmetric control transfer is limited to separate stacks, which makes some symmetric patterns too expensive in terms of consumed memory.
- In most cases a stackful coroutine cannot be moved safely between system threads because its execution context may contain invocations of non-coroutine functions which rely on thread-specific information like locks and thread variables.

7. Acknowledgements

The authors would like to thank Anton Stoyanov, Aleksandar Cheshmedjiev, Alexander Penev, Aleksandar Tanchev, Asen Krastev, Georgi L Dimitrov, Gor Nishanov, Hristosko Chaushev, Lewis Baker, Peter Dimov, Richard Smith, Stanimir Lukanov and Yordan Sarachinov for comments on early drafts of this proposal.

Thanks again to all those involved in the Coroutines TS, and particularly Gor Nishanov, for exploring the coroutines design space and providing a basis for this paper.

References

[1] Coroutines TS, N4760, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4760.pdf
[2] Core Coroutines, P1063R1, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1063r1.pdf
[3] Resumable expressions, P0114R0, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0114r0.pdf
[4] fiber_handle - fibers without scheduler, P0876R3, http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p0876r3.pdf

Appendix: Examples

Following are several code examples which demonstrate how our design would be used to solve some more realistic tasks.

Base class for generators implemented as coroutines

The challenge when implementing a generator as a coroutine is that the generator needs to be able to complete without producing any values. In our design coroutines can only complete by executing a type-safe `return` statement which produces the same value type as `yield()` does.

The following implementation addresses this by providing a general-purpose base class for generators that derives from `coroutine<void(void)>` and overrides `yield()` so that it returns a value. As the coroutine body needs to be able to return to the same caller through both `yield()` and `return`, we introduce an intermediate coroutine `_start`, which stores its caller in an explicit continuation to allow `yield` to return to the consumer.

In order for asynchronous generators to work correctly, both `done` and `next` are coroutines too.

```
template<typename Val>
class generator : protected coroutine<void(void)> {
   // If the optional is set, a value was produced but not yet consumed
   optional<Val> _val;

   // Used to allow both start and yield to transfer control to advance.
   resume_continuation<void(void)> _cont;

   // An intermediate coroutine used to store the caller of the generator.
   coroutine<void(void)> start_coro(generator& self) {
      // Store the caller in _cont to allow yield() to pass control to it.
      self._cont = get_caller();

      // Start the body of the generator. It will keep calling the yield
      // member coroutine and won't return here until the sequence is
      // complete.
      self();

      // At this point the generator has executed a return statement.
      self._cont();
   }

   start_coro _start;

   // If necessary, advances the generator by one step.
   coroutine<void(void)> advance_coro(generator& self) {
      while (!self.coroutine<void(void)>::done()) {
         // Only advance if the last value was consumed.
         if (!self._val.has_value())
            self._start();

         // The first time start is invoked, it will move its caller
         // continuation to _cont. When yield invokes _cont, control
         // will return here and the continuation inside start will be
         // updated. The next invocation of start will return control
         // to yield.

         // Give control back to done or next.
         yield();
      }

      // After the generator body is complete, just yield back to the caller
      for(;;)
         yield();
   }

   advance_coro _advance;

protected:
   // The yield member coroutine overrides the standard yield() with one
   // that takes a value.
   coroutine<void(Val& val)> yield_coro(generator& self) {
      // Save the first generated value
      self._val = get_initial_value();

      for (;;) {
         // Return control to advance
         self._cont();

         // Return control to the generator body to produce the next value
         self._val = yield();
      }
   }
   yield_coro yield;

public:
   // Implements the logic of done.
   coroutine<bool(void)> done_coro(generator& self) {
      for (;;) {
         self._advance();
         yield(self.coroutine<void(void)>::done());
      }
   }

   done_coro done;

   // Implements the logic of next.
   coroutine<Val()> next_coro(generator& self) {
      for (;;) {
         self._advance();

         // Reset the optional to indicate that advance needs to call
         // the generator again.
         optional<Val> val;
         std::swap(val, self._val);
         yield(*val);
      }
   }

   next_coro next;

   generator()
      : _start(*this), _advance(*this), yield(*this)
      , done(*this) , next(*this)
   {}
};

generator<int> range(int start, int end) {
   for (int i = start; i < end; ++i)
      yield(i);
}

int main() {
   range r(0, 10);

   while (!r.done())
      cout << r.next() << endl;
}
```

A naive regex matcher

Here we present a simple implementation of a recursive descent matcher for regular expressions.

We represent a regex pattern as a factory function which creates a coroutine that finds all matches of the pattern at a specific position in the input. Patterns can be combined to create more complex patterns.

The matcher coroutines need to invoke recursively other coroutines. These coroutines need to be allocated on the heap. To simplify the creation and management of the recursive coroutines we introduce a wrapper coroutine which can be allocated on the stack.

This example is analogous in many ways to the Parser Combinator example from the Core Coroutines paper.

```
using position = string::size_type;

struct Input {
   string& text;
   position pos;

   Input(string& s, size_t pos) : text(s), pos(pos) {}
};

// A wrapper to allow to store matcher coroutines on the stack
// without slicing. An extended version of this should end up
// in the standard library
template<typename Wrapped, typename... Args>
coroutine<position(void)> wrap(Args&&... args) {
   auto wrapped = make_unique<Wrapped>(std::forward<Args>(args)...);

   while (true) {
      position result = (*wrapped)();
      if (wrapped->done())
         return result;
      yield(result);
   }
}

// A pattern is a factory, which takes a position and returns a wrapped
// coroutine  that tries to match the pattern at this position
using Pattern = std::function<wrap(Input)>;
```

```
coroutine<position(void)> prim_coro(Input input, string pattern) {
   if (input.text.compare(input.pos,pattern.size(), pattern) == 0)
      return (input.pos + pattern.size());
   else
      return string::npos;
}

Pattern prim(const string& pattern) {
   return [=](Input input) { return wrap<prim_coro>(input, pattern); };
}

// This coroutine matches sequences
coroutine<position(void)> seq_coro(Input input, Pattern p1, Pattern p2) {
   // For all positions that match the first pattern, match the second one
   auto p1_coro = p1(input);
   while (!p1_coro.done()) {
      position pos1 = p1_coro();

      if (pos1 == string::npos)
        return string::npos;

      auto p2_coro = p2( Input{input.text, pos1} );
      while (!p2_coro.done()) {
         position pos2 = p2_coro();
         if (pos2 == string::npos)
            break;
         yield(pos2)
      }
   }

   return string::npos;
}

Pattern seq(Pattern p1, Pattern p2) {
   return [p1, p2](Input input) { return wrap<seq_coro>(input, p1, p2); };
}

// This coroutine matches alternatives
coroutine<position(void)> alt_coro(Input input, Pattern p1, Pattern p2) {
   // Yield all matches of the first alternative, than all of the other
   auto p1_coro = p1(input);
   for (position pos = p1_coro(); !p1_coro.done(); pos = p1_coro())
      yield(pos);

   auto p2_coro = p2(input);
   for (position pos = p2_coro(); !p2_coro.done(); pos = p2_coro())
      yield(pos);
   return string::npos;
}

Pattern alt(Pattern p1, Pattern p2) {
   return [p1, p2](Input input) { return wrap<alt_coro>(input, p1, p2); };
}

// This coroutine matches repetitions
Pattern rep(Pattern p);

coroutine<position(void)> rep_coro(Input input, Pattern p) {
   // For all positions that match the pattern, create the same pattern
   // and call it recursively.
   auto head_coro = p(input);
   Pattern tail = rep(p);

   while (!head.done()) {
      position pos1 = head();

      if (pos == string::npos)
         return string::npos;

      auto tail_coro = tail(Input{input.text, pos1});
      while (!tail_coro.done()) {
         position pos2 = tail_coro();
         if (pos2 == string::npos)
            break;
         yield(pos2);
      }
   }

   return string::npos;
}

Pattern rep(Pattern p) {
   return [p](Input input) { return wrap<rep_coro>(input, p); };
}

// A helper function to find the longest match
position match_longest(Pattern& p, const string& input) {
   auto matcher = p( Input{string, 0}; );
   position longest = std::string::npos;

   while (!matcher.done())
      longest = max(result, matcher());
   return longest;
}

int main() {
   // (aaaa|aa)(bbbb|bb)
   auto pattern = seq(alt(prim("aaaa"), prim("aa")), alt(prim("bbbb"),
prim("bb")));

   if (match_longest(pattern, "aabb ") == 4) cout << "OK" << endl;
   if (match_longest(pattern, "aaaabb ") == 6) cout << "OK" << endl;
   if (match_longest(pattern, "aabbbb ") == 6) cout << "OK" << endl;
   if (match_longest(pattern, "aaaabbbb ") == 8) cout << "OK" << endl;

   return 0;
}
```

Simulating the Coroutines TS design with our proposal

Following is an implementation of the most important features of the Coroutines TS. The implementation of co_await demonstrates how the fully symmetric coroutines in our design can be used in place of awaiters. It also demonstrates the use of detach() to break the invocation chain and the use of get_root_coroutine(). The implementation of co_yield demonstrates how a coroutine member variable can be used as a resumable method of a class.

```
// Type helpers
template<class Sig> struct mem_fun_types;

template<class C, class R, class Arg>
struct mem_fun_types<R(C::*)(Arg)> {
   using arg_type = Arg;
   using ret_type = R;
};

template<class C, class R>
struct mem_fun_types<R(C::*)()> {
   using arg_type = void;
   using ret_type = R;
};

template<class Awaitable>
struct co_await_type {
   using fp_type = decltype(&Awaitable::await_resume);
   using type = typename mem_fun_types<fp_type>::ret_type;
};
```

```
template<class Promise>
struct yield_type {
   using fp_type = decltype(&Promise::yield_value);
   using value_type = typename mem_fun_types<fp_type>::arg_type;
};

// A coroutine class that implements the semantics of the Coroutines TS
template<typename Wrapper>
class ts_coroutine : public coroutine<void(void), void(void)> {
   resume_continuation<void(void), void(void)> resumer;

public:
   using Promise = coroutine_traits<Wrapper>::promise_type;

   class handle {
      friend class ts_coroutine;
      handle(ts_coroutine* instance) : instance(instance) {}
      ts_coroutine* instance;

   public:
      handle(const handle other) : instance(other.instance) {}

      Promise* get_promise() { return &(instance->promise); }
      void operator()() { instance->resumer(); }
      void resume() { instance->resumer(); }
      bool done() const { return instance->resumer.done(); }
      void destroy() { delete instance;}
   };

protected:
   // A helper for await
   template<class Awaitable>
   coroutine<void(void), void(void)>
   suspend(ts_coroutine* self, Awaitable& expr) {
      // Store the resume continuation of the calling await
      // in the ts_coroutine object
      self->resumer = get_caller();

      // Ask the awaitable for the next coroutine to resume
      handle next = expr.await_suspend(handle(self));

      // Resume the next coroutine. This will return to the
      // corresponding await.
      if (next.instance != nullptr)
         detach(next.instance.resume_continuation);
      else
```

```
        detach(get_root_coroutine());
    }

    // A coroutine that implements the generic co_await algorithms
    template<Awaitable>
    coroutine<co_await_type<Awaitable>::type(Awaitable), void(void)>
    await(ts_coroutine* self) {
        Awaitable expr = get_initial_value();

        if (!expr.await_ready())
            suspend(self, expr)();

        return expr.await_resume();
    }

    // This is a factory function that creates the co_await context
    template<Awaitable>
    auto co_await() {
        return await<Awaitable>(*this);
    }

    // A coroutine that implements co_yield. Assumes that co_yield doesn't
    // return a value.
    using yield_value_type = yield_type<Promise>::value_type;

    coroutine<void(yield_value_type), void(void)>
    yielder(ts_coroutine* self) {
        auto value_to_yield = get_initial_value();

        while (true) {
            co_await()(self->promise.yield_value(value_to_yield));
            value_to_yield = yield();
        }
    }

    yielder co_yield;

    template<typename... Args>
    ts_coroutine(Args&&... args)
         : promise(std::forward<Args>(args)...)
         , co_yield(this)
         , body(*this)
    {}

private:
    // Implement a coroutine that will wrap the user supplied one
    coroutine<void(void), void(void)> Body(ts_coroutine& self) {
        co_await()(self.promise.initial_suspend());

        // This doesn’t strictly work, because our design doesn’t specify yet
        // how exceptions will be propagated. We will add exceptions support
        // in the near future.
        try {
           self();
        } catch(...) {
           self.promise.unhandled_exception(get_current_exception());
        }

        co_await()(self.promise.final_suspend());
    }

    Promise promise;
    Body body;

public:
    // A type-erasing factory that creates a TS coroutine on the heap
    // and starts it
    template<typename C, typename... Args>
    static Wrapper factory(Args&&... args) {
        C* c = new C(std::forward<Args>(args)...);
        handle h(c);

        Wrapper wrapper = c->promise.get_return_object();
        c.body();
        return wrapper;
    }
};

using coroutine_handle = ts_coroutine::handle;

ts_coroutine<generator<int>> counter() {
    for (int i = 0; ; ++i)
        co_yield(i);
}

int main() {
    generator<int> gen = counter::factory<counter>();
}
```