Document Number: N4244

Date: 2014-10-13

Reply To Christopher Kohlhoff <chris@kohlhoff.com>

Resumable Lambdas:

A language extension for generators and coroutines

1 Introduction

N3977 *Resumable Functions* and successor describe a language extension for resumable functions, or "stackless" coroutines. The earlier revision, N3858, states that the motivation is:

> *[...] the increasing importance of efficiently handling I/O in its various forms and the complexities programmers are faced with using existing language features and existing libraries.*

In contrast to "stackful" coroutines, a prime use case for stackless coroutines is in server applications that handle many thousands or millions of clients. This is due to stackless coroutines having lower memory requirements than stackful coroutines, as the latter, like true threads, must have a reserved stack space. Memory cost per byte is dropping over time, but with server applications at this scale, per-machine costs, such as hardware, rack space, network connectivity and administration, are significant. For a given request volume, it is always cheaper if the application can be run on fewer machines.

Existing best practice for stackless coroutines in C and C++ uses macros and a `switch`-based technique similar to Duff's Device. Examples include the `coroutine` class included in Boost.Asio[1], and Protothreads[2]. Simon Tatham's web page[3], *Coroutines in C*, inspired both of these implementations. A typical coroutine using this model looks like this:

```
void operator()(error_code ec, size_t n)
{
  if (!ec) reenter (this)
  {
    for (;;)
    {
      yield socket_->async_read_some(buffer(*data_), *this);
      yield async_write(*socket_, buffer(*data_, n), *this);
    }
  }
}
```

The memory overhead of a stackless coroutine can be as low as a single `int`. The downside is that, being stackless, the developer is of course unable to use stack-based variables. Instead, care must be taken to use only data in long lasting memory locations, such as class members.

Thus, the author fully agrees that it would be beneficial to have true language support for stackless coroutines, in order to address these shortcomings. Unfortunately, the language extensions presented in N3977 include design choices that prevent them from attaining an equivalent level of efficiency.

[1] http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio/overview/core/coroutine.html

[2] http://dunkels.com/adam/pt/

[3] http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html

In Rapperswil, SG1 asked the authors of N3977 resumable functions and N3985 stackful coroutines to unify the proposals as far as possible. Interestingly, at the time of this request the authors of the proposals themselves were strongly against such a course of action. This author agrees with their position and feels that, at least partly, SG1's request was a consequence of N3977 resumable functions not offering a sufficiently distinct value proposition. Stackless coroutines are worthy of being considered as an independent feature, but in doing so should make full use of C++'s support for lightweight, value-based types.

In this paper we will introduce a potential language extension that has the same minimal runtime overhead as the macro-based approach. While the implementation is based on prior art in pure C++, this proposal also draws design inspiration from Python, a programming language with rich experience in generators and coroutines.

2 How lightweight can we get?

As a simple example, let's consider a generator function object. When implemented with Boost.Asio's `coroutine` class and its supporting macros `reenter` and `yield`, it might look like this:

```
struct generator
{
  coroutine coro;
  int n, m;
  std::string s, t;
  int operator()()
  {
    reenter (coro)
    {
      for (n = 0; n != 10; ++n)
      {
        if (n % 2)
        {
          m = n * 3;
          yield return m;
        }
        else
        {
          s = std::string(n, 'x');
          t = std::string(n, 'y');
          yield return s.size() + t.size();
        }
      }
    }
    assert(0 && "got to end");
  }
};
```

Here, `sizeof(generator)` is the sum of the sizes of `coroutine` (which contains only an `int`), two `int`s and two `std::string` objects. Furthermore, we can embed objects of type `generator` as a data member of other objects.

For a slight improvement in brevity, we can also express the generator as a lambda:

```
auto generator = [coro = coroutine(), n = int(), m = int(),
    s = std::string(), t = std::string()]() mutable -> int
{
  reenter (coro)
  {
    for (n = 0; n != 10; ++n)
    {
      if (n % 2)
      {
        m = n * 3;
        yield return m;
      }
      else
      {
        s = std::string(n, 'x');
        t = std::string(n, 'y');
        yield return s.size() + t.size();
      }
    }
  }
  assert(0 && "got to end");
}
```

This will have the same size as the `struct`-based generator above. However, if we want to embed the generator within another object, we must be able to name the type. We can do that by wrapping the lambda in a function:

```
inline auto generator()
{
  return [coro = coroutine(), n = int(), m = int(),
      s = std::string(), t = std::string()]() mutable -> int
  {
    // ...
  };
}

typedef decltype(generator()) generator_t;
```

In both cases above the compiler is able to inline calls to the generator, ensuring that the mechanism has little overhead when compared to a hand-rolled function object. Furthermore, the object may be arbitrarily copied, as with any other value type.

3 Introducing resumable lambdas

Macro-based stackless coroutines give full control over where the data is stored, but at some small cost in brevity and convenience. We have to specify the variables as data members in a handcrafted function object, or in a lambda capture.

What we really want is something that is as lightweight as a handcrafted function object, but with the syntactic convenience of N3977 resumable functions.

The second example in section 2 points the way: *resumable lambdas*. The basis for this approach is as follows:

- The body of a lambda is available to the compiler at the point where the lambda type is defined.
- The compiler can analyse the lambda body to determine what stack variables need to be accommodated.
- Space can be made available for the stack variables in exactly the same way as is already done for the capture set.

Or, put another way:

`sizeof(`*lambda*`)` == `sizeof(`*capture-set*`)` + `sizeof(`*stack-vars*`)` + `sizeof(`*resume-point*`)`

The generator in section 2 can then be rewritten with the aid of two new keywords, `resumable` and `yield`.

```
[]() resumable -> int
{
  for (int n = 0; n != 10; ++n)
  {
    if (n % 2)
    {
      int m = n * 3;
      yield m;
    }
    else
    {
      std::string s(n, 'x');
      std::string t(n, 'y');
      yield s.size() + t.size();
    }
  }
}
```

4 Writing simple generators

Consider a simple generator that counts down from a specified starting value:

```
auto g = [n = int(10)]() resumable
{
  std::cout << "Counting down from " << n << "\n";
  while (n > 0)
  {
    yield n;
    n -= 1;
  }
};
```

Instead of returning a single value as in a normal lambda, we generate a sequence of values using the `yield` statement. To obtain these values we call the generator as a normal function object:

```
try
{
  for (;;)
    std::cout << g() << "\n";
}
catch (std::stop_iteration&)
{
}
```

If we fall off the end of the generator body, which happens here when `n == 0`, the generator automatically throws an exception of type `std::stop_iteration`. However, in many use cases an exception will not be acceptable. To avoid the exception we need to prevent the generator from falling off the end. One approach is to return a special value:

```
auto g = [n = int(10)]() resumable
{
  std::cout << "Counting down from " << n << "\n";
  while (n > 0)
  {
    yield n;
    n -= 1;
  }
  return -1;
};

for (int n = g(); n != -1; n = g())
  std::cout << n << "\n";
```

Another approach is to ensure that the last value returned by the generator uses `return` rather than `yield`. This will transition the generator into the terminal state, which is something we can test for.

```
auto g = [n = int(10)]() resumable
{
  std::cout << "Counting down from " << n << "\n";
  while (n > 0)
  {
    if (n == 1)
      return n;
    yield n;
    n -= 1;
  }
};

while (!g.is_terminal())
  std::cout << g() << "\n";
```

5 Using a resumable lambda as a data member

We can store our resumable lambda as a data member within another class. This is achieved by wrapping the resumable lambda in a factory function. For our countdown generator above, we would write this as follows:

```
auto countdown(int n)
{
  return [n]() resumable
  {
    std::cout << "Counting down from " << n << "\n";
    while (n > 0)
    {
      yield n;
      n -= 1;
    }
  };
}

typedef decltype(countdown(0)) countdown_t;

class printer
{
public:
  explicit printer(int n) : countdown_(countdown(n)) {}
  // ...
private:
  countdown_t countdown_;
};
```

By embedding resumable lambdas inside classes, we can use them to implement state machine-like behaviour. We also have the ability to group related resumable lambdas together for improved locality of reference.

6 Copying and moving resumable lambdas

Like regular lambdas, resumable lambdas have an implicitly declared copy constructor, and an implicitly declared move constructor. This allows us to copy them freely like other value-based types, with the additional behaviour that when we copy them we also copy the `yield` point at which they are currently suspended. For example:

```
auto g1 = []() resumable
{
  yield 1;
  yield 2;
  yield 3;
  yield 4;
};

std::cout << "g1 returned " << g1() << "\n";
```

```
std::cout << "g1 returned " << g1() << "\n";
auto g2 = g1;
std::cout << "g1 returned " << g1() << "\n";
std::cout << "g2 returned " << g2() << "\n";
```

will print:

```
g1 returned 1
g1 returned 2
g1 returned 3
g2 returned 3
```

Unlike macro-based stackless coroutines, resumable lambdas allow stack variables within the lambda body. For example:

```
template <int N> struct copyable
{
  copyable() {}
  copyable(const copyable&) { std::cout << "copying " << N << "\n"; }
};

auto g1 = []() resumable
{
  {
    copyable<1> c1;
    yield 1;
    yield 2;
  }
  {
    copyable<2> c2;
    yield 3;
    yield 4;
  }
};

std::cout << "g1 returned " << g1() << "\n";
std::cout << "g1 returned " << g1() << "\n";
auto g2 = g1; // g1 is currently suspended immediately after "yield 2;"
std::cout << "g1 returned " << g1() << "\n";
std::cout << "g2 returned " << g2() << "\n";
```

will print:

```
g1 returned 1
g1 returned 2
copying 1
g1 returned 3
g2 returned 3
```

as we copy the resumable lambda while it is suspended immediately after "`yield 2;`".

However, automatically copying stack variables means that the presence of a non-copyable stack variable must inhibit the generation of the copy constructor. Similarly, a non-movable stack variable inhibits the move constructor.

Needless to say, a lambda object that can be neither copied nor moved is of limited utility. At an absolute minimum, we need to be able to copy or move a newly constructed lambda object into a durable memory location.

Unfortunately, at the time of writing the author does not have a good general solution to this problem. The easiest solution would be to generate the copy and move constructors but fail at runtime unless `is_initial()` is true, but this was rejected as potentially introducing unintended behaviour.

7 Supporting asynchronous operations – part I

Next, let us turn our attention to the composition of asynchronous operations using resumable lambdas. Consider the Boost.Asio stackless coroutine example from the Introduction in its complete form:

```
struct server_loop : coroutine
{
  tcp::socket* socket_;
  vector<char>* data_;

  void operator()(error_code ec, size_t n)
  {
    if (!ec) reenter (this)
    {
      for (;;)
      {
        yield socket_->async_read_some(buffer(*data_), *this);
        yield async_write(*socket_, buffer(*data_, n), *this);
      }
    }
  }
};
```

The `server_loop` object footprint is extremely small: just one `int` and two pointers. As we start a new each asynchronous operation, the function object is passed by value, copied and stored by the library implementation until the operation completes.

If we want do the equivalent with resumable lambdas, we immediately encounter a problem: we have no way to access the lambda's own `this` pointer. Therefore, this paper proposes a language extension to enable this access: `[]this`. That is, a *lambda-introducer* containing no *lambda-capture*, followed by the `this` keyword.

With this facility, we can now write the resumable lambda like so:

```
[socket, data](error_code ec, size_t n) resumable
{
  while (!ec)
  {
    yield socket->async_read_some(buffer(*data), *[]this);
    if (!ec)
      yield async_write(*socket, buffer(*data, n), *[]this);
  }
}
```

The generated code is virtually identical to the macro-based stackless coroutine version above.

8 Stacking resumable lambdas – part I

A resumable lambda may delegate to another generator using a `yield from` expression. In this case, the expression to the right of `yield from` must evaluate to an object that is a generator (i.e. it meets the Generator type requirements).

For example, if we delegate to the countdown generator we defined earlier, we can perform two consecutive counts as follows:

```
auto g = []() resumable
{
  yield from countdown(3);
  yield from countdown(2);
};

try
```

```
{
  for (;;)
    std::cout << g() << “\n”;
}
catch (std::stop_iteration&)
{
}
```

This will print:

```
3
2
1
2
1
```

In this example, a `yield from` expression is roughly equivalent to:

```
auto tmp = countdown(3);
while (!tmp.is_terminal())
  return tmp();
```

However, `yield from` may also be used as a sub-expression:

```
int result = yield from countdown(3);
```

In this case, only the final value yielded by `countdown` will be assigned into `result`. These semantics are intended to support asynchronous operations, as we will see below. Note that when `yield from` is used as sub-expression, the return type of the enclosing resumable lambda must either be default constructible or `void`.

9 Using yield as an expression

To allow resumable lambdas to be used as coroutines, the `yield` keyword may be used as an expression. In this form it must be followed by a *type-id,* using syntax similar to `sizeof`. For example:

```
auto f = []() resumable
{
  int n = yield(int);
  std::cout << "Received " << n << "\n";
  std::string s = yield(std::string);
  std::cout << "Received " << s << "\n";
};
```

This allows us to send values into the resumable lambda. We first prime the lambda by calling it as a function object:

```
f();
```

It is now suspended at the `yield` expression. We supply the value by using the `wanted` member function, and then resume the lambda.

```
*f.wanted<int>() = 42;
f();
*f.wanted<std::string>() = "hello";
f();
```

It will now print:

```
received 42
received hello
```

We can also determine the type being waited for at runtime by calling the `wanted_type` member function:

```
const std::type_info& type = c.wanted_type();
```

This allows resumable lambdas to safely wait for different types at different suspension points.

10 Stacking resumable lambdas – part II

When we stack resumable lambdas using `yield from`, the `wanted` function is automatically forwarded to the inner generator. For example, given two generators:

```
auto f1()
{
  return []() resumable
  {
    std::cout << "f1 received " << yield(int) << "\n";
    std::cout << "f1 received " << yield(int) << "\n";
  };
}

auto f2()
{
  return []() resumable
  {
    std::cout << "f2 received " << yield(int) << "\n";
    std::cout << "f2 received " << yield(int) << "\n";
  };
}
```

and a resumable lambda that composes them:

```
auto f3 = []() resumable
{
  yield from f1();
  yield from f2();
};
```

the following sequence of calls:

```
f3(); // primes the resumable lambda
for (int i = 0; i < 4; ++i)
{
  *f3.wanted<int>() = i;
  f3();
}
```

will print:

```
f1 received 0
f1 received 1
f2 received 2
f2 received 3
```

11 Supporting asynchronous operations – part II

In the asynchronous operations example above, all operations were required to have compatible callback signatures. By combining the ability to stack lambdas and `yield` expressions, we are now able to support the composition of dissimilar asynchronous operations. We do this by having our asynchronous operations return a generator.

Given a resumable lambda:

```
[]() resumable
{
  ...
  size_t length = yield from async_read_2(socket, buffer, *[]this);
  ...
}
```

The `async_read_2` function may be implemented in terms of callback-based `async_read` like this:

```
struct async_read_generator // implements Generator requirements
{
  struct result
  {
    enum { initial, ready, terminal } state_ = ready;
    error_code ec_;
    size_t length_ = 0;
  } result_;

  size_t operator()()
  {
    if (state_ == result::ready) state_ = result::terminal;
    if (result_.ec_) throw system_error(ec);
    return result_.length_;
  }

  const std::type_info& wanted_type() const noexcept
  {
    return typeid(result);
  }

  template <class T> T* wanted() noexcept
  {
    if (!is_same<T, result>::value && !is_same<T, void>::value) return nullptr;
    return static_cast<T*>(static_cast<void*>(&result_));
  }

  template <class T> const T* wanted() const noexcept
  {
    if (!is_same<T, result>::value && !is_same<T, void>::value) return nullptr;
    return static_cast<const T*>(static_cast<const void*>(&result_));
  }

  bool is_terminal() const noexcept
  {
    return result_.state == result::terminal;
  }
};

template <class Resumable>
async_read_generator async_read_2(Socket socket, Buffer buffer, Resumable r)
{
  async_read(socket, buffer,
    [r=std::move(r)](error_code ec, size_t length)
    {
      r.wanted<async_read_generator::result>()->ec_ = ec;
      r.wanted<async_read_generator::result>()->length_ = length;
      r.wanted<async_read_generator::result>()->state_ = async_read_generator::result::ready;
      r();
    });
  return async_read_generator();
}
```

It is not intended that this wrapper be hand-crafted for every asynchronous operation. Instead, the extensible asynchronous model described in N4045 *Library Foundations for Asynchronous Operations* may be used to create the necessary generators automatically.

12 Summary of proposed changes

12.1 Language changes

Resumable lambdas are supported by the following additions to the grammar:

- `resumable`
  - Used to denote a resumable lambda, in the same way that `mutable` is used to denote a lambda that may modify captured variables.
- `yield ;`
  - Used in the same places where `return ;` may appear.
  - Yields control of the coroutine to the caller.
- `yield` *unary-expression* `;`
  - Used in the same places where `return` *unary-expression* `;` may appear.
- `yield (` *type-id* `)`
  - May be used either as a return-like expression, or as an rvalue within another expression.
  - The type specified by type-id must be DefaultConstructible.
- `yield from` *unary-expression*
  - The type of *unary-expression* must satisfy the Generator requirements.
  - May be used either as a return-like expression, or as an rvalue within another expression.
- `[]this`
  - Used to obtain the resumable lambda's own `this` pointer, as distinct from a captured `this`.

To avoid needlessly breaking existing code, it is suggested that the `yield` and `from` keywords be context sensitive, i.e. that they only be treated as keywords if they appear within a lambda that is marked as `resumable`.

12.2 Resumable lambda classes

From the point of view of a user, a resumable lambda is equivalent to a class defined as follows:

```
struct __resumable_lambda
{
  // Always generated.
  __resumable_lambda();
  ~__resumable_lambda();

  // Only generated if all capture set members and stack variables are copyable.
  __resumable_lambda(const __resumable_lambda& other);

  // Only generated if all capture set members and stack variables are movable.
  __resumable_lambda(__resumable_lambda&& other);

  // Return type and arguments determined by lambda call signature.
  R operator()(Args...);

  // Always generated.
  const std::type_info& wanted_type() const noexcept;
  template <class T> T* wanted () noexcept;
  template <class T> const T* wanted() const noexcept;
  bool is_initial() const noexcept;
  bool is_terminal() const noexcept;
};
```

12.3 Standard library support

Resumable lambdas require only the following addition to the standard library.

```
class stop_iteration : exception { ... };
```

12.4 Generator type requirements

In a `yield from` *unary-expression*, the type `G` of the unary expression must satisfy the Generator requirements specified in the table below. Let `a` be an lvalue of type `G`, `b` be a const lvalue of type `G`, and `T` be an arbitrary type.

Expression	**Type**	**Notes**
`a()`	Any type.	Requires: `!is_terminal()`. Returns the next available value from the generator.
`b.is_terminal()`	`bool`	Shall not throw an exception. Indicates that the generator has reached a terminal state and will produce no further values.
`b.wanted_type()`	`const std::type_info&`	Shall not throw an exception. Returns a `type_info` object that identifies the type that the generator is currently waiting for. Returns `typeid(void)` if the generator is not waiting for a value.
`a.wanted<T>()`	`T*`	Shall not throw an exception. If the generator is not waiting for a value, returns `nullptr`. If `T` is `void`, or `typeid(T) == a.wanted_type()`, returns a pointer to the wanted value. Otherwise returns `nullptr`.
`b.wanted<T>()`	`const T*`	Shall not throw an exception. If the generator is not waiting for a value, returns `nullptr`. If `T` is `void`, or `typeid(T) == b.wanted_type()`, returns a pointer to the wanted value. Otherwise returns `nullptr`.

13 Implementation approach

13.1 Simple generators

An implementation of resumable lambdas bears similarity to both of the following situations:

- Destruction of stack-based variables when unwinding block scopes (whether due to normal return or due to an exception).
- Destruction of data members within a partially constructed aggregate.

With resumable lambdas, the main additional consideration is the existence of yield points. We will begin by taking the example generator from section 2 and giving each unwind point[4] and yield point a unique integer label. For demonstration purposes, the `int` variables are labelled as unwind points even though this is not strictly necessary.

```
[]() resumable -> int /* __state = 0 */
{
  for (int n = 0 /* __state = 1 */; n != 10; ++n)
  {
    if (n % 2)
    {
      int m = n * 3; /* __state = 2 */
      yield m; /* __state = 3 */
    }
    else
    {
      std::string s(n, 'x'); /* __state = 4 */
      std::string t(n, 'y'); /* __state = 5 */
      yield s.size() + t.size(); /* __state = 6 */
    }
  }
} /* __state = ~0 */
```

Our resumable lambda is then mapped to a function object as follows:

```
struct __resumable_lambda0
{
```

The currently active unwind point or yield point is stored as an integer.

```
  int __state;
```

Every block scope is represented by a struct. The struct names are derived from the unwind/yield point that is active at the end of the block. Where a block scope contains nested scopes, these are contained within a union.

```
  union
  {
    struct
    {
      int n;
      union
      {
        struct
        {
          int m;
        } __s3;
        struct
        {
          std::string s;
          std::string t;
        } __s6;
      };
    } __s1;
  };
```

The constructor of the object sets the state to an initial value that indicates that execution has not yet started, and no stack variables have been initialised.

```
  __resumable_lambda0() : __state(0) {}
```

The destructor must be able to destroy the variables that are currently alive at any of the unwind or yield points.

[4] An "unwind point" determines which stack variables must be destroyed were an exception thrown at that point.

```
~__resumable_lambda0() { __destructor(); }
void __destructor()
{
  switch (__state)
  {
  // __s6
  case 6: __state6:
  case 5: __state5:
    { typedef std::string __type; __s1.__s6.t.~__type(); }
  case 4: __state4:
    { typedef std::string __type; __s1.__s6.s.~__type(); }
    goto __state1;
  // __s3
  case 3: __state3:
  case 2: __state2:
    // __s1.__s3.m.~int();
    goto __state1;
  // __s1
  case 1: __state1:
    // __s1.n.~int();
    goto __state0;
  // __s0
  case 0: case ~0: __state0:
    break;
  default:
    std::terminate();
  }
  __state = 0;
}
```

On the other hand, copy construction is only possible when the resumable lambda is in its initial or terminal state, or is suspended at a yield point.

```
__resumable_lambda0(const __resumable_lambda0& __other)
{
  try
  {
    switch (__other.__state)
    {
    case 0: case ~0:
      break;
    case 3: case 6: // __s1
      new (&__s1.n) int(__other.__s1.n), __state = 1;
      switch (__other.__state)
      {
      case 3: // __s3
        new (&__s1.__s3.m) int(__other.__s1.__s3.m), __state = 3;
        break;
      case 6:
        new (&__s1.__s6.s) std::string(__other.__s1.__s6.s), __state = 4;
        new (&__s1.__s6.t) std::string(__other.__s1.__s6.t), __state = 5;
        break;
      }
      break;
    default:
      std::terminate();
    }
  }
  catch (...)
  {
    __destructor();
    throw;
  }
  __state = __other.__state;
}
```

The implementation of `operator()` is switch-based, just as in the macro approach.

```
  int operator()()
  {
    switch (__state)
    {
    case 0: (void)0;
      for (new (&__s1.n) int(0), __state = 1; __s1.n != 10; ++__s1.n)
      {
        if (__s1.n % 2)
        {
          new (&__s1.__s3.m) int(__s1.n * 3), __state = 2;
          __state = 3; return __s1.__s3.m; case 3: (void)0;
          /* __s1.__s3.m.~int(); */ __state = 1;
        }
        else
        {
          new (&__s1.__s6.s) std::string(__s1.n, 'x'), __state = 4;
          new (&__s1.__s6.t) std::string(__s1.n, 'y'), __state = 5;
          __state = 6; return __s1.__s6.s.size() + __s1.__s6.t.size(); case 6: (void)0;
          { typedef std::string __type; __s1.__s6.t.~__type(), __state = 4; }
          { typedef std::string __type; __s1.__s6.s.~__type(), __state = 1; }
        }
      }
      /* __s1.n.~int(); */ __state = 0;
      __state = ~0; case ~0: throw std::stop_iteration();
    }
  }
```

Finally, to round out the class, we provide some member functions to query a resumable lambda's current state.

```
  bool is_initial() const noexcept { return __state == 0; }
  bool is_terminal() const noexcept { return __state == ~0; }
};
```

13.2 `yield` (*type-id*) and `wanted`

To see `yield ( ` *type-id* ` )` in action, let us consider how a simple resumable lambda would map to an equivalent function object.

```
[]() resumable
{
  for (;;)
  {
    int i = yield(int);
    std::cout << i << std::endl;
    int j = yield(int);
    std::cout << j << std::endl;
    std::string k = yield(std::string);
    std::cout << k << std::endl;
  }
}
```

When a resumable lambda object is "blocked" on a `yield` expression, the `wanted` member function is used to supply the pending value.

```
auto f = ...;
f();
*f.wanted<int>() = 1;
f(); // writes 1 to std::cout
*f.wanted<int>() = 2;
f(); // writes 2 to std::cout
*f.wanted<std::string>() = "abc";
f(); // writes abc to std::cout
```

To implement `yield` and `wanted`, we will begin by identifying the integer labels needed for the yield and unwind points.

```
[]() resumable /* __state = 0 */
{
  for (;;)
  {
    int i = yield(int); /* __state = 1 : waiting for value */
                        /* __state = 2 : i has been constructed */
    std::cout << i << std::endl;
    int j = yield(int); /* __state = 3 : waiting for value */
                        /* __state = 4 : j has been constructed */
    std::cout << j << std::endl;
    std::string k = yield(std::string); /* __state = 5 : waiting for value */
                                        /* __state = 6 : k has been constructed */
    std::cout << k << std::endl;
  }
} /* __state = ~0 */
```

The expression `yield(int)` also tells the compiler that it needs storage for a "temporary" object of type `int`. In addition to the block scope containing the named variables, each temporary resides in its own block scope.

```
struct __resumable_lambda1
{
  int __state;
  union
  {
    struct
    {
      union
      {
        struct
        {
          int __tmp1;
        } __s1;
        struct
        {
          int __tmp3;
        } __s3;
        struct
        {
          std::string __tmp5;
        } __s5;
      };
      int i;
      int j;
      std::string k;
    } __s6;
  };
```

The implementation of `operator()` is similar in approach to the earlier `yield`-based example.

```
  void operator()()
  {
    switch (__state)
    {
    case 0: (void)0;
      for (;;)
      {
        __state = 1; return; case 1: (void)0;
        new (&__s6.i) (std::move(__s6.__s1.__tmp1));
        /* __s6.__s1.__tmp1.~int(); */ __state = 2;
        std::cout << i << std::endl;
        __state = 3; return; case 3: (void)0;
        new (&__s6.j) (std::move(__s6.__s3.__tmp3));
        /* __s6.__s3.__tmp3.~int(); */ __state = 4;
        std::cout << j << std::endl;
        new (&__s6.__s5.__tmp5) std::string();
        __state = 5; return; case 5: (void)0;
        new (&__s6.k) std::string(std::move(__s6.__s5.__tmp5));
        { typedef std::string __type; __s6.__s5.__tmp5.~__type(), __state = 6; }
        std::cout << k << std::endl;
        { typedef std::string __type; __s6.k.~__type(), __state = 4; }
        /* __s6.j.~int(); */ __state = 2;
        /* __s6.i.~int(); */ __state = 0;
      }
      __state = ~0; case ~0: throw std::stop_iteration();
    }
  }
```

The wanted member function returns the correct “waiting” variable depending on the current state.

```
  template <class T> __wanted_tag {};

  template <class T > T* wanted()
  {
    void* __p = this->__wanted(__wanted_tag<T>());
    return static_cast<T*>(__p);
  }

  void* wanted(__wanted_tag<int>)
  {
    switch (__state)
    {
    case 1: return &__s6.__s1.__tmp1;
    case 3: return &__s6.__s3.__tmp3;
    default: return nullptr;
    }
  }

  void* __wanted(__wanted_tag<std::string>)
  {
    switch (__state)
    {
    case 5: return &__s6.__s5.__tmp5;
    default: return nullptr;
    }
  }

  void* __wanted(__wanted_tag<void>)
  {
    switch (__state)
    {
    case 1: return &__s6.__s1.__tmp1;
    case 3: return &__s6.__s3.__tmp3;
    case 5: return &__s6.__s5.__tmp5;
    default: return nullptr;
    }
  }

  template <class T> void* __wanted(__wanted_tag<T>)
  {
    return nullptr;
  }
```

14 Future work

The author is currently developing a prototype of this language feature using a Clang-based pre-processor.

15 Conclusion

Stackless coroutines provide an elegant solution to many problems including, but importantly not limited to, composition of asynchronous operations. Their key benefits over stackful coroutines are low memory usage, and the ability to be copied, moved and stored like any other value type.

The N3977 resumable functions deny programmers these benefits. However, language extensions still have value in making it easier to write stackless coroutines. In this paper we have introduced an alternate approach, resumable lambdas, that can give the same syntactic benefits, but with the low overhead of value-based function objects.

16 Acknowledgements

Thanks go to Jamie Allsop and Richard Smith for providing comments on early drafts of this paper.

17 Appendix: Design issues in N3977 and successor

17.1 Unspecified memory representation

N3977's problems stem from the fact that a resumable function's associated memory representation is hidden from the user. The earlier revision of N3977, N3858, suggests two implementation approaches: side stacks, and heap-allocated frames.

17.1.1 Side stacks as a worst case

When using a side stack, the implementation is effectively a stackful coroutine. Stackful coroutines have inherently greater memory consumption, so this is a standard library quality-of-implementation issue. Therefore, when developing portable code, the user cannot assume that they will be able to efficiently handle the "millions of clients" scenario.

Worse, while bearing the cost of stackful coroutines, the user is unable to take advantage of their benefits, such as the ability to suspend the coroutine from a nested function call. The user would be better off simply using library-based stackful coroutines, such as those presented in N3985. These offer the same opportunity for readable asynchronous code, but without the need for a language extension.

17.1.2 Heap allocated frames aren't much better

The memory layout within a heap-allocated frame is more-or-less equivalent to that used by the macro-based best practice. However, in N3977 the user is not given the opportunity to specify where the frame is stored.

The successor to N3977 proposes the use of a `resumable_traits<>` type to customise resumable function behaviour, and includes an allocator type. This improves the situation, but does not provide a convenient solution for passing context on a per-allocation basis.

As a result, the user cannot easily take advantage of locality of reference using N3977 resumable functions. By contrast, resumable lambdas are normal function objects. Like other objects, related coroutines can be stored close together. We can use them as class members.

17.2 Coupling to futures and promises

N3977 specifies that resumable functions must return either `std::future` or `std::shared_future`. As illustrated in N4045 *Library Foundation for Asynchronous Operations*, the use of `std::future` introduces inherent costs, such as synchronisation, and inhibits optimisation by preventing inlining. These costs add up as we use resumable functions to add layers of abstraction.

The coupling to `std::future` also prevents the use of resumable lambdas as generator functions. N3858 does describe a `sequence<>` type that could be used for this. However, this type appears to use runtime polymorphism. This, in conjunction with the unspecified memory representation, means that it can never be as efficient as a hand-rolled function object.

The successor to N3977 partially addresses this by utilising a promise/future-based *concept*, rather than being tied to specific type. However, this author believes that this concept is more complex than it needs to be. The generator type requirements described in this proposal offer a simpler, more flexible model.

17.3 More complex standard library support

The successor to N3977 requires non-trivial support from the standard library, in the form of `resumable_traits<>` and `resumable_handle`. The library support required for resumable lambdas consists only of the `std::stop_iteration` exception.