Await 2.0
Stackless Resumable Function

MOST SCALABLE, MOST EFFICIENT, MOST OPEN COROUTINES OF ANY PROGRAMMING LANGUAGE IN EXISTENCE

CppCon 2014 • Gor Nishanov (gorn@microsoft.com) • Microsoft

What this talk is about

- Evolution of N3858 and N3977
- Stackless Resumable Functions (D4134)
  - Lightweight, customizable coroutines
  - Proposed for C++17
  - Experimental implementation “to be” released in Visual Studio “14”
- What are they?
- How they work?
- How to use them?
- How to customize them?

Coroutines

56 years ago

- Introduced in 1958 by Melvin Conway

- Donald Knuth, 1968: “generalization of subroutine”

	subroutines	coroutines
call	Allocate frame, pass parameters	Allocate frame, pass parameters
return	Free frame, return result	Free frame, return eventual result
suspend	x	yes
resume	x	yes

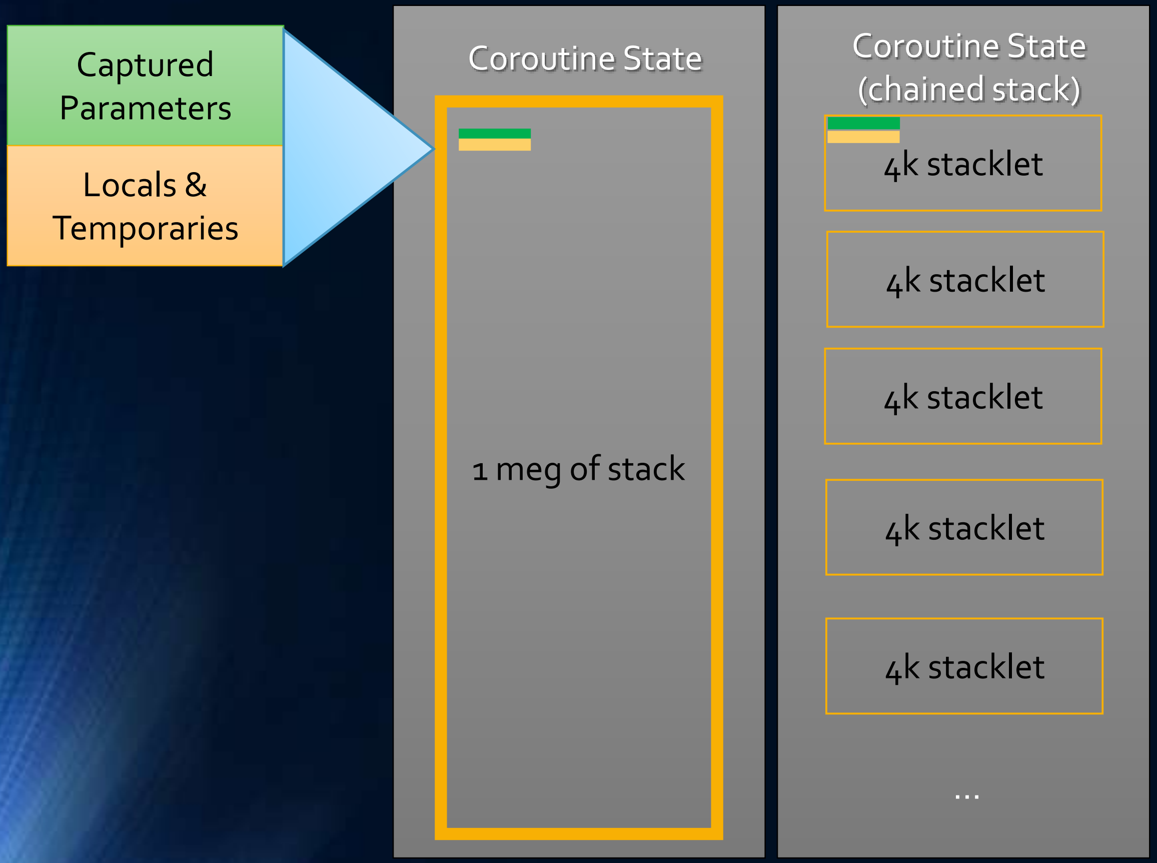
User Mode Threads / Fibers

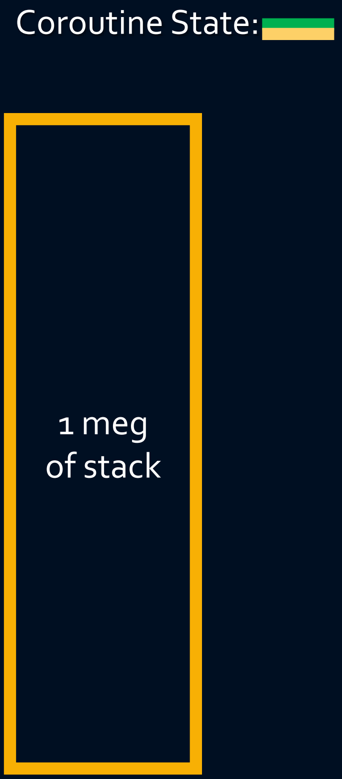
Stackless Resumable Functions

Coroutine classification

- Symmetric / Asymmetric
  - Modula-2 / Win32 Fibers / Boost::context are symmetric (SwitchToFiber)
  - C# asymmetric (distinct suspend and resume operations)
- First-class / Constrained
  - Can coroutine be passed as a parameter, returned from a function, stored in a data structure?
- Stackful / Stackless
  - How much state coroutine has? Just the locals of the coroutine or entire stack?
  - Can coroutine be suspended from nested stack frames

Stackful vs. Stackless





Design Goals

- Highly scalable (to hundred millions of concurrent coroutines)
- Highly efficient (resume and suspend operations comparable in cost to a function call overhead)
- Seamless interaction with existing facilities **with no overhead**
- Open ended coroutine machinery allowing library designers to develop coroutine libraries exposing various high-level semantics, such as generators, goroutines, tasks and more.
- Usable in environments where exception are forbidden or not available

Anatomy of a Function

```
std::future<ptrdiff_t> tcp_reader(int total)
{
    char buf[64 * 1024];
    ptrdiff_t result = 0;

    auto conn =










}
```

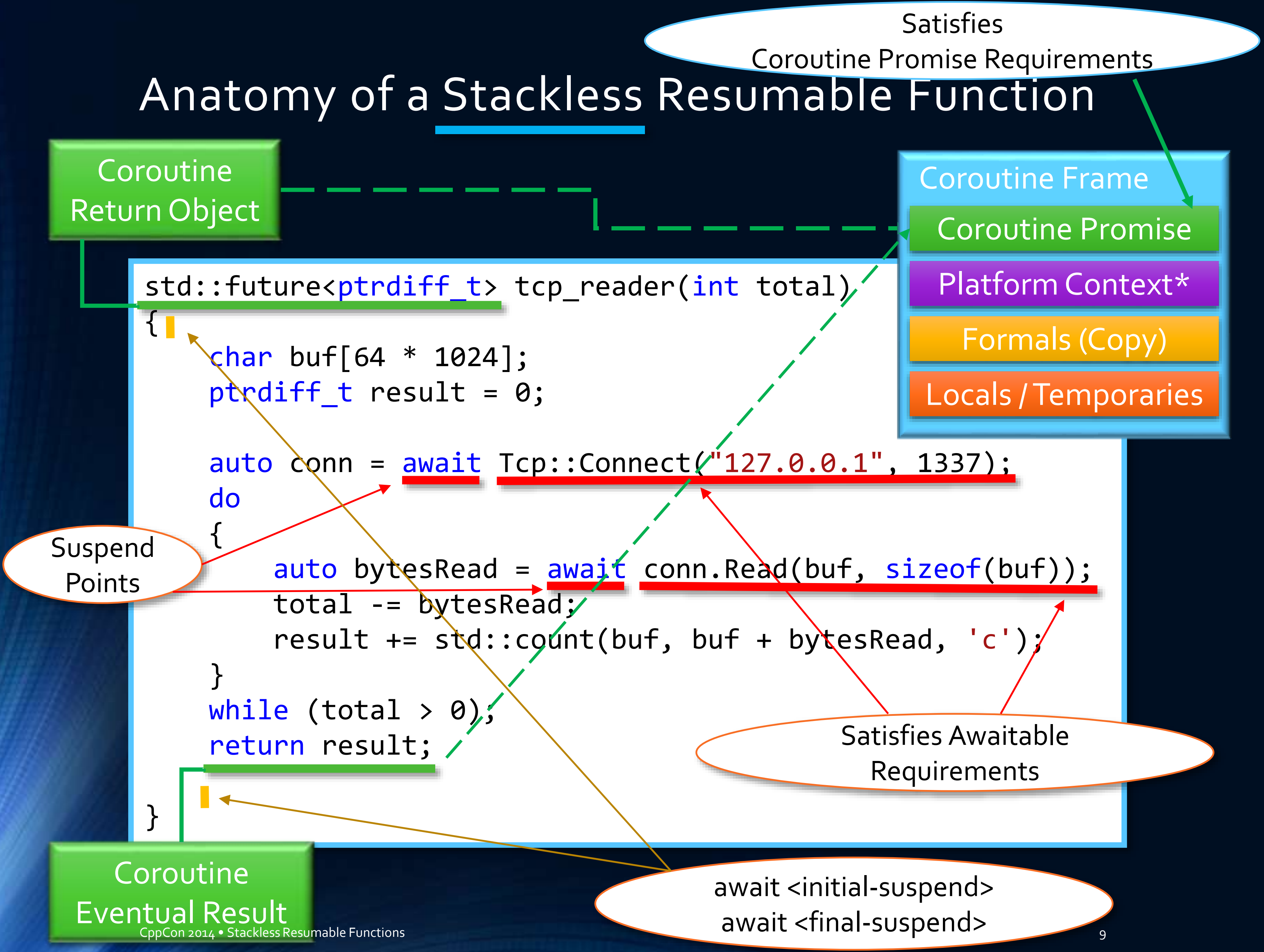
Anatomy of a Resumable Function

```
std::future<ptrdiff_t> tcp_reader(int total)
{
    char buf[64 * 1024];
    ptrdiff_t result = 0;

    auto conn = await Tcp::Connect("127.0.0.1", 1337);
    do
    {
        auto bytesRead = await conn.Read(buf, sizeof(buf));
        total -= bytesRead;
        result += std::count(buf, buf + bytesRead, 'c');
    }
    while (total > 0);
    return result;

}
```

Anatomy of a Stackless Resumable Function



```
std::future<ptrdiff_t> tcp_reader(int total)
{
    char buf[64 * 1024];
    ptrdiff_t result = 0;

    auto conn = await Tcp::Connect("127.0.0.1", 1337);
    do
    {
        auto bytesRead = await conn.Read(buf, sizeof(buf));
        total -= bytesRead;
        result += std::count(buf, buf + bytesRead, 'c');
    }
    while (total > 0);
    return result;

}
```

Satisfies
Coroutine Promise Requirements

Coroutine Return Object

Coroutine Frame

Coroutine Promise

Platform Context*

Formals (Copy)

Locals / Temporaries

Suspend
Points

Satisfies Awaitable
Requirements

Coroutine
Eventual Result

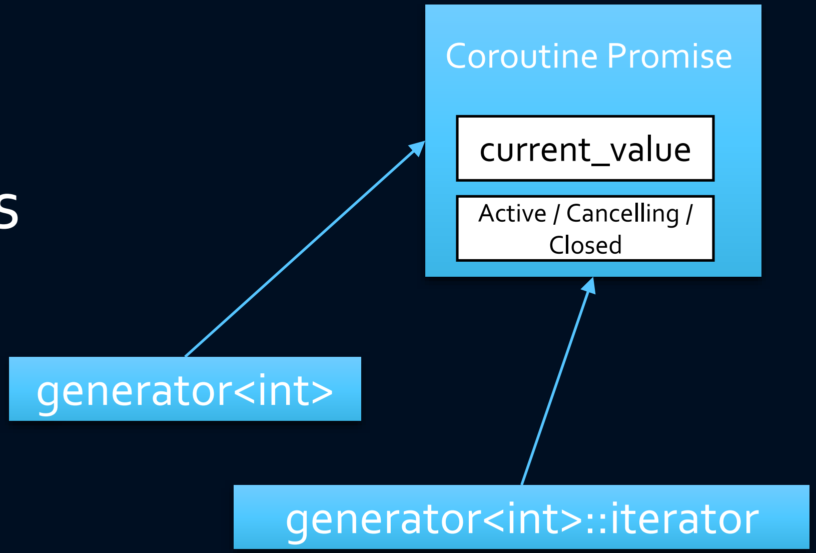
await <initial-suspend>
await <final-suspend>

2 x 2 x 2

- Two new keywords
  - **await**
  - **yield**
- Two new concepts
  - Awaitable
  - Coroutine Promise
- Two new types
  - resumable_handle
  - resumable_traits

Examples

Generator coroutines



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

```
int main() {
   for (auto v : fib(35))
   {
      if (v > 10)
          break;
      cout << v << ' ';
   }
}
```

```
{
   auto && __range = fib(35);
   for (auto __begin = __range.begin(),
             __end = __range.end()
       ;
       __begin != __end
       ;
       ++__begin)
   {
      auto v = *__begin;
      {
         if (v > 10) break;
         cout << v << ' ';
      }
   }
}
```

Recursive Generators

```
recursive_generator<int> range(int a, int b)
{
    auto n = b - a;

    if (n <= 0)
        return;

    if (n == 1)
    {
        yield a;
        return;
    }

    auto mid = a + n / 2;

    yield range(a, mid);
    yield range(mid, b);
}
```

```
int main()
{
    auto r = range(0, 100);
    copy(begin(r), end(r),
           ostream_iterator<int>(cout, " "));
}
```

Parent-stealing scheduling

```
spawnable<int> fib(int n) {
    if (n < 2) return n;
    return await(fib(n - 1) + fib(n - 2));
}

int main() { std::cout << fib(5).get() << std::endl; }
```

1,4 billion recursive invocations to compute fib(43), uses less than 16k of space
Not using parent-stealing, runs out of memory at fib(35)

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

Goroutines?

```
goroutine pusher(channel<int>& left, channel<int>& right) {
   for (;;) {
      auto val = await left.pull();
      await right.push(val + 1);
   }
}
```

Goroutines? Sure. 100,000,000 of them

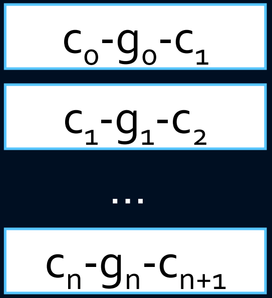
```
goroutine pusher(channel<int>& left, channel<int>& right) {
   for (;;) {
      auto val = await left.pull();
      await right.push(val + 1);
   }
}
```

```
int main() {
   const int N = 100 * 1000 * 1000;
   vector<channel<int>> c(N + 1);

   for (int i = 0; i < N; ++i)
      goroutine::go(pusher(c[i], c[i + 1]));

   c.front().sync_push(0);

   cout << c.back().sync_pull() << endl;
}
```



c_0-g_0-c_1

c_1-g_1-c_2

…

c_n-g_n-c_{n+1}

Reminder: Just Core Language Evolution

Library Designer Paradise

- Lib devs can design new coroutines types
  - generator<T>
  - goroutine
  - spawnable<T>
  - task<T>
  - …
- Or adapt to existing async facilities
  - std::future<T>
  - concurrency::task<T>
  - IAsyncAction, IAsyncOperation<T>
  - …

Awaitable

Reminder: Range-Based For

```
int main() {
   for (auto v : fib(35))
      cout << v << endl;
}
```

```
{
   auto && __range = fib(35);
   for (auto __begin = __range.begin(),
             __end = __range.end()
       ;
       __begin != __end
       ;
       ++__begin)
   {
      auto v = *__begin;
      cout << v << endl;
   }
}
```

await <expr>

Expands into expression equivalent of

If <expr> is a class type and unqualified ids await_ready, await_suspend or await_resume are found in the scope of a class

```
{
  auto && __tmp = <expr>;
  if (!__tmp.await_ready()) {
      __tmp.await_suspend(<resumption-function-object>);
  }
  <cancel-check>
  return __tmp.await_resume();
}
```

suspend

resume

await <expr>

Expands into expression equivalent of

Otherwise
(see rules for range-based-for
lookup)

```
{
  auto && __tmp = <expr>;
  if (! await_ready(__tmp)) {
      await_suspend(__tmp, <resumption-function-object>);
  }
  <cancel-check>
  return await_resume(__tmp);
}
```

suspend

resume

Trivial Awaitable #1

```
struct _____blank____ {
   bool await_ready(){ return false; }
   template <typename F>
   void await_suspend(F const&){}
   void await_resume(){}
};
```

Trivial Awaitable #1

```
struct suspend_always {
   bool await_ready(){ return false; }
   template <typename F>
   void await_suspend(F const&){}
   void await_resume(){}
};
```

```
await suspend_always {};
```

Trivial Awaitable #2

```cpp
struct suspend_never {
   bool await_ready(){ return true; }
   template <typename F>
   void await_suspend(F const&){}
   void await_resume(){}
};
```

Simple Awaitable #1

```
void DoSomething(mutex& m) {
   unique_lock<mutex> lock = await lock_or_suspend{m};
   // ...
}
```

```
struct lock_or_suspend {
   std::unique_lock<std::mutex> lock;
   lock_or_suspend(std::mutex & mut) : lock(mut, std::try_to_lock) {}

   bool await_ready() { return lock.owns_lock(); }

   template <typename F>
   void await_suspend(F cb)
   {
      std::thread t([this, cb]{ lock.lock(); cb(); });
      t.detach();
   }

   auto await_resume() { return std::move(lock);}
};
```

Simple Awaiter #2: Making Boost.Future awaitable

```
#include <boost/thread/future.hpp>
namespace boost {

   template <class T>
   bool await_ready(unique_future<T> & t) {
     return t.is_ready();
   }

   template <class T, class F>
   void await_suspend(unique_future<T> & t,
                      F resume_callback)
   {
       t.then([=](auto&){resume_callback();});
   }

   template <class T>
   auto await_resume(unique_future<T> & t) {
      return t.get(); }
   }
}
```

Awaitable
Interacting with C APIs

2 x 2 x 2

- Two new keywords
  - **await**
  - **yield**
- Two new concepts
  - Awaitable
  - Coroutine Promise
- Two new types
  - resumable_handle
  - resumable_traits

resumable_handle

```
template <typename Promise = void> struct resumable_handle;

template <> struct resumable_handle<void> {
   void operator() ();
   void * to_address();
   static resumable_handle<void> from_address(void*);
…
};

template <typename Promise>
struct resumable_handle: public resumable_handle<> {
   Promise & promise();
   static resumable_handle<Promise> from_promise(Promise*);
   …
};
```

```
== != < > <= >=
```

Simple Awaitable #2: Raw OS APIs

```
await sleep_for(10ms);
```

```
class sleep_for {
    static void TimerCallback(PTP_CALLBACK_INSTANCE, void* Context, PTP_TIMER) {
       std::resumable_handle<>::from_address(Context)();
    }
    PTP_TIMER timer = nullptr;
    std::chrono::system_clock::duration duration;
public:
    sleep_for(std::chrono::system_clock::duration d) : duration(d){}

    bool await_ready() const { return duration.count() <= 0; }

    void await_suspend(std::resumable_handle<> resume_cb)  {
       int64_t relative_count = -duration.count();
       timer = CreateThreadpoolTimer(TimerCallback, resume_cb.to_address(), 0);
       SetThreadpoolTimer(timer, (PFILETIME)&relative_count, 0, 0);
    }

    void await_resume() {}

    ~sleep_for() { if (timer) CloseThreadpoolTimer(timer); }
};
```

2 x 2 x 2

- Two new keywords
  - **await**
  - **yield**
- Two new concepts
  - Awaitable
  - Coroutine Promise
- Two new types
  - resumable_handle
  - resumable_traits

resumable_traits

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

```
std::resumable_traits<generator<int>, int>
```

```
template <typename R, typename... Ts>
struct resumable_traits {
   using allocator_type = std::allocator<char>;
   using promise_type = typename R::promise_type;
};
```

Defining Coroutine Promise for boost::future

```
namespace std {
  template <typename T, typename… anything>
  struct resumable_traits<boost::unique_future<T>, anything…> {
     struct promise_type {
        boost::promise<T> promise;
        auto get_return_object() { return promise.get_future(); }

        template <class U> void set_value(U && value) {
            promise.set_value(std::forward<U>(value));
        }

        void set_exception(std::exception_ptr e) {
           promise.set_exception(std::move(e));
        }
        suspend_never initial_suspend() { return{}; }
        suspend_never final_suspend() { return{}; }

        bool cancel_requested() { return false; }
    };
};
```

Awaitable and Exceptions

Exceptionless Error Propagation (Await Part)

```
#include <boost/thread/future.hpp>

namespace boost {

   template <class T>
   bool await_ready(unique_future<T> & t) { return t.is_ready();}

   template <class T, class F>
   void await_suspend(
      unique_future<T> & t, F rh)
   {
       t.then([=](auto& result){
          rh();
       });
   }



   template <class T>
   auto await_resume(unique_future<T> & t) { return t.get(); }
}
```

Exceptionless Error Propagation (Await Part)

```
#include <boost/thread/future.hpp>

namespace boost {

   template <class T>
   bool await_ready(unique_future<T> & t) { return t.is_ready();}

   template <class T, class Promise>
   void await_suspend(
      unique_future<T> & t, std::resumable_handle<Promise> rh)
   {
       t.then([=](auto& result){
          if(result.has_exception())
            rh.promise().set_exception(result.get_exception_ptr());
          rh();
       });
   }

   template <class T>
   auto await_resume(unique_future<T> & t) { return t.get(); }
}
```

Exceptionless Error Propagation (Promise Part)

```
namespace std {
  template <typename T, typename… anything>
  struct resumable_traits<boost::unique_future<T>, anything…> {
     struct promise_type {
        boost::promise<T> promise;

        auto get_return_object() { return promise.get_future(); }

        suspend_never initial_suspend() { return{}; }
        suspend_never final_suspend() { return{}; }

        template <class U> void set_value(U && value) {
            promise.set_value(std::forward<U>(value));
        }

        void set_exception(std::exception_ptr e) {
           promise.set_exception(std::move(e));
        }
        bool cancel_requested() { return false; }
    };
};
```

Exceptionless Error Propagation (Promise Part)

```
namespace std {
  template <typename T, typename… anything>
  struct resumable_traits<boost::unique_future<T>, anything…> {
     struct promise_type {
        boost::promise<T> promise;

        auto get_return_object() { return promise.get_future(); }

        suspend_never initial_suspend() { return{}; }
        suspend_never final_suspend() { return{}; }

        template <class U> void set_value(U && value) {
            promise.set_value(std::forward<U>(value));
        }

        void set_exception(std::exception_ptr e) {
           promise.set_exception(std::move(e));
        }
        bool cancel_requested() { return promise.has_error(); }
    };
};
```

Simple Happy path and reasonable error propagation

```
std::future<ptrdiff_t> tcp_reader(int total)
{
    char buf[64 * 1024];
    ptrdiff_t result = 0;

    auto conn = await Tcp::Connect("127.0.0.1", 1337);
    do
    {
        auto bytesRead = await conn.Read(buf, sizeof(buf));
        total -= bytesRead;
        result += std::count(buf, buf + bytesRead, 'c');
    }
    while (total > 0);
    return result;

}
```

await <expr>

Expands into expression equivalent of

```
{
  auto && __tmp = <expr>;
  if (! await_ready(__tmp)) {
      await_suspend(__tmp, <resumption-function-object>);
                                                              suspend
                                                              resume
  }
  if (<promise>.cancellation_requested()) goto <end-label>;
  return await_resume(__tmp);
}
```

Done!

What this talk was about

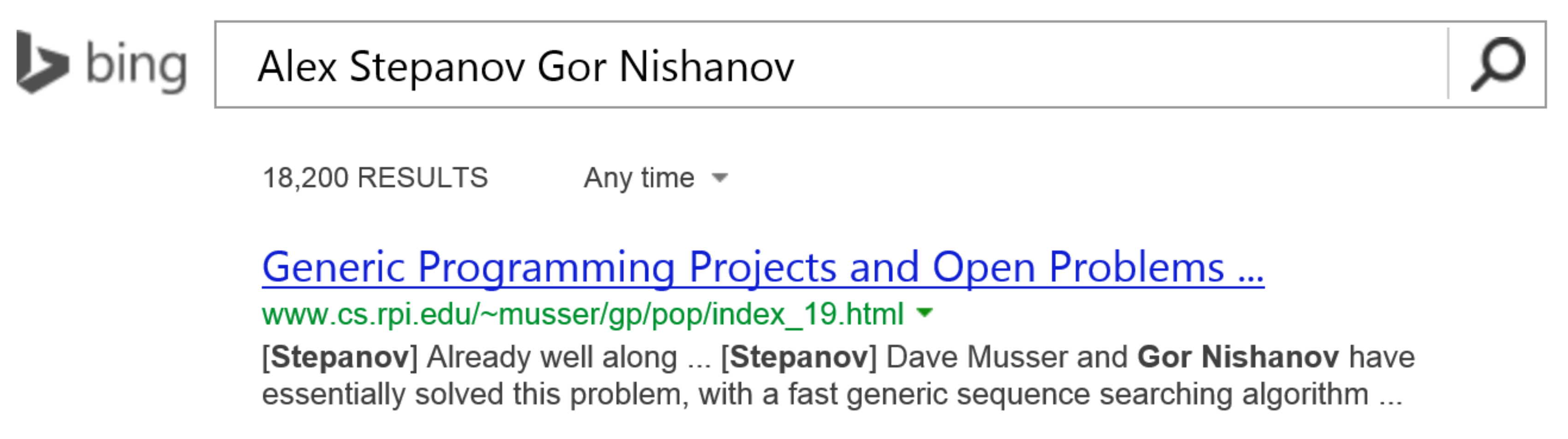
- Stackless Resumable Functions (D4134)
  - Lightweight, customizable coroutines
  - Proposed for C++17
  - Experimental implementation “to be” released in Visual Studio “14”
- What are they?
- How they work?
- How to use them?
- How to customize them?

To learn more:

- https://github.com/GorNishanov/await/
  - Draft snapshot: D4134 Resumable Functions v2.pdf
- In October 2014 look for
  - N4134 at http://isocpp.org
  - http://open-std.org/JTC1/SC22/WG21/

Backup

Introduction



Alex Stepanov Gor Nishanov

18,200 RESULTS   Any time

Generic Programming Projects and Open Problems ...

www.cs.rpi.edu/~musser/gp/pop/index_19.html

[**Stepanov**] Already well along ... [**Stepanov**] Dave Musser and **Gor Nishanov** have essentially solved this problem, with a fast generic sequence searching algorithm ...

How does it work?

Generator coroutines

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

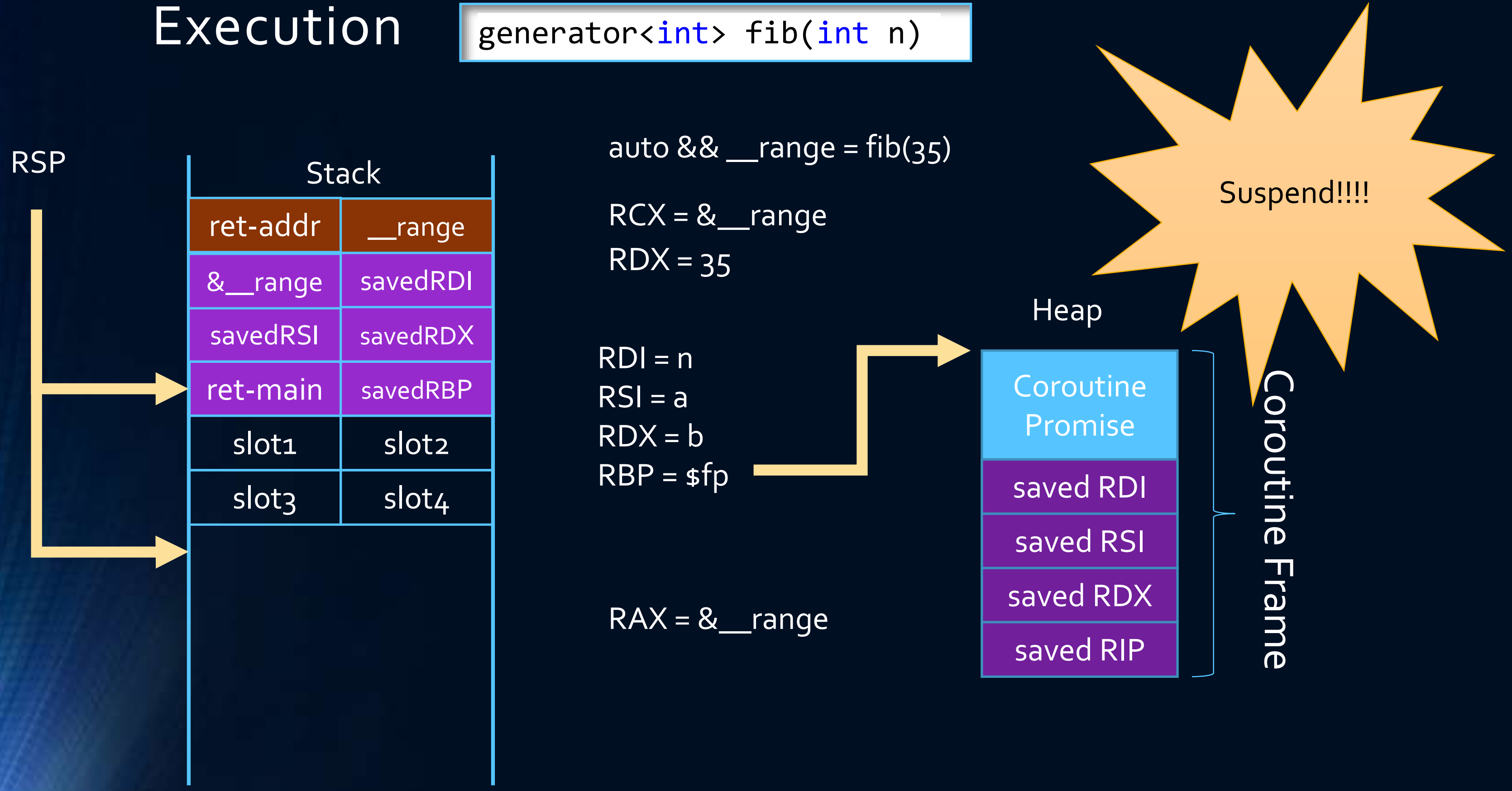
```
int main() {
   for (auto v : fib(35))
      cout << v << endl;
}
```

```
{
   auto && __range = fib(35);
   for (auto __begin = __range.begin(),
             __end = __range.end()
       ;
       __begin != __end
       ;
       ++__begin)
   {
      auto v = *__begin;
      cout << v << endl;
   }
}
```

x86_x64 Windows ABI

Execution

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



auto && __range = fib(35)

RCX = &__range
RDX = 35

RDI = n
RSI = a
RDX = b
RBP = $fp

RAX = &__range

Stack

ret-addr	__range
&__range	savedRDI
savedRSI	savedRDX
ret-main	savedRBP
slot1	slot2
slot3	slot4

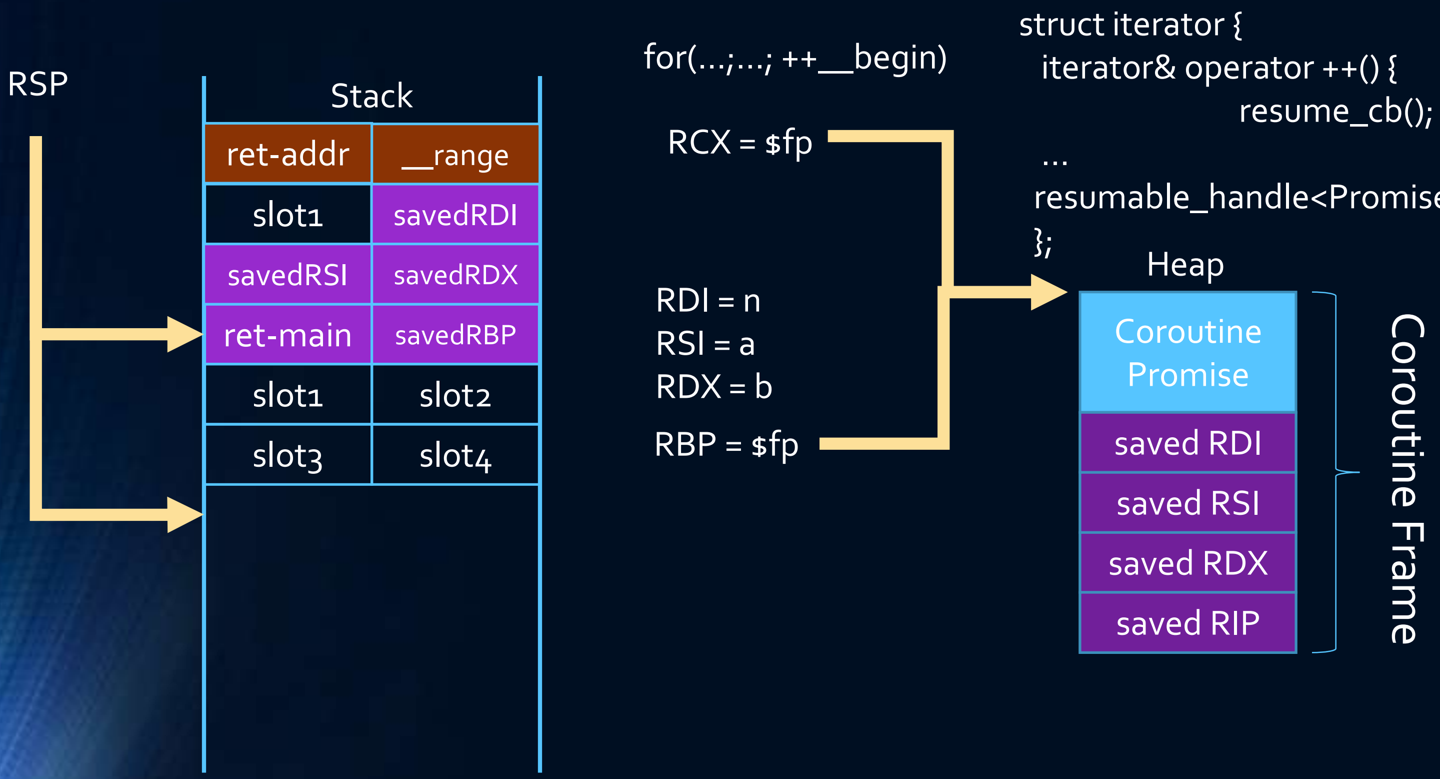
Heap

- Coroutine Promise
- saved RDI
- saved RSI
- saved RDX
- saved RIP

Resume

```
generator<int>::iterator::operator ++()
```

```
struct iterator {
  iterator& operator ++() {
                    resume_cb();  return *this; }
  …
 resumable_handle<Promise> resume_cb;
};
```



ret-addr	__range
slot1	savedRDI
savedRSI	savedRDX
ret-main	savedRBP
slot1	slot2
slot3	slot4

Coroutine Promise Requirement

```
return <expr>
```

→

```
<Promise>.set_value(<expr>);
 goto <end>
```

<unhandled-exception> →

```
<Promise>.set_exception (
          std::current_exception())
```

<get-return-object> →

```
<Promise>.get_return_object()
```

```
yield <expr>
```

→

```
await <Promise>.yield_value(<expr>)
```

<before-last-curly> →

```
await <Promise>.initial_suspend()
```

<after-first-curly> →

```
await <Promise>.final_suspend()
```

<cancel-check> →

```
if(<Promise>.cancellation_requested())
   <goto end>
```

If await_suspend returns bool

await <expr>

Expands into expression equivalent of

```
{
  auto && __tmp = <expr>;
  if (! await_ready(__tmp) &&
      await_suspend(__tmp, <resumption-function-object>) {
```

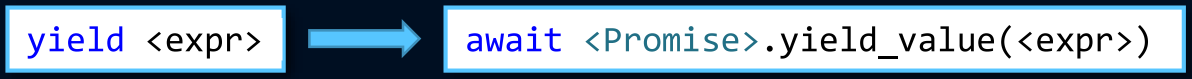
suspend

resume

```
  }
  if (<promise>.cancellation_requested()) goto <end-label>;
  return await_resume(__tmp);
}
```

Yield implementation

compiler:



```
yield <expr>
```

```
await <Promise>.yield_value(<expr>)
```

library:

```
suspend_now
generator<T>::promise_type::yield_value(T const& expr) {
   this->current_value = &expr;
   return{};
}
```

awaitable_overlapped_base

```
struct awaitable_overlapped_base : public OVERLAPPED
{
    ULONG IoResult;
    ULONG_PTR NumberOfBytesTransferred;
    std::resumable_handle<> resume;

    static void __stdcall io_complete_callback( PTP_CALLBACK_INSTANCE,
                PVOID, PVOID Overlapped, ULONG IoResult,
                ULONG_PTR NumberOfBytesTransferred,
                PTP_IO)
    {
        auto o = reinterpret_cast<OVERLAPPED*>(Overlapped);
        auto me = static_cast<awaitable_overlapped_base*>(o);

        me->IoResult = IoResult;
        me->NumberOfBytesTransferred = NumberOfBytesTransferred;
        me->resume();
    }
};
```

Dial awaitable

```
class Dial : public awaitable_overlapped_base {
    ports::endpoint remote;
    Connection conn;
public:
    Dial(string_view str, unsigned short port) : remote(str, port) {}
    bool await_ready() const { return false; }
    void await_suspend(std::resumable_handle<> cb)  {
       resume = cb;
       conn.handle = detail::TcpSocket::Create();
       detail::TcpSocket::Bind(conn.handle, ports::endpoint("0.0.0.0"));
       conn.io = CreateThreadpoolIo(conn.handle, &io_complete_callback, 0,0);
       if (conn.io == nullptr)  throw_error(GetLastError());

          StartThreadpoolIo(conn.io);
       auto error = detail::TcpSocket::Connect(conn.handle, remote, this);
       if (error) { CancelThreadpoolIo(conn.io); throw_error(GetLastError());
   }
   Connection await_resume() {
       if (conn.error)  throw_error(error);
       return std::move(conn); }
};
```

Connection::Read

```
auto Connection::read(void* buf, size_t bytes) {
   class awaiter : public awaitable_overlapped_base {
       void* buf; size_t size;
       Connection * conn;
   public:
       awaiter(void* b, size_t n, Connection * c): buf(b), size(n), conn(c) {}
       bool await_ready() const { return false; }
       void await_suspend(std::resumable_handle<> cb)  {
           resume = cb;
           StartThreadpoolIo(conn->io);
           auto error = TcpSocket::Read(conn->handle, buf, (uint32_t)size, this);
           if (error)
              { CancelThreadpoolIo(conn->io); throw_error(error); }
       }
       int await_resume() {
           if (IoResult)
              { throw_error(IoResult); }
           return (int)this->NumberOfBytesTransferred; }
   };
   return awaiter{ buf, bytes, this };
}
```

asynchronous iterator helper: await for

```
goroutine foo(channel<int> & input) {
   await for(auto && i : input) {
      cout << “got: “ << i << endl;
   }
}
```

await for expands into:

```
{
      auto && __range = range-init;
      for ( auto __begin = await (begin-expr),
            __end = end-expr;
            __begin != __end;
            await ++__begin )
      {
            for-range-declaration = *__begin;
            statement
      }
}
```

Recursive Tree Walk (Stackful)

```
void traverse ( node_t * n, std::push_coroutine<std::string> & yield) {
    if(n-> left ) traverse (n->left, yield);
    yield (n-> value);
    if(n-> right ) traverse (n->right, yield);
}
node * root1 = create_tree();
node * root2 = create_tree();

std::pull_coroutine<std::string> reader1( [&](auto & yield ){ traverse (root1, yield);});
std::pull_coroutine<std::string> reader2( [&](auto & yield ){ traverse (root2, yield);});

std :: cout << “equal = “ << std::equal (begin (reader1), end( reader1), begin(reader2))
                 << std :: endl ;
```

Recursive Tree Walk (Stackless)

```
generator<std::string> traverse(node_t* n)
{
    if (p->left) yield traverse(p->left);
    yield p->name;
    if (p->right) yield traverse(p->right);
}

node * root1 = create_tree();
node * root2 = create_tree();

auto reader1 = traverse (root1);
auto reader2 = traverse (root2);

std :: cout << “equal = “ << std::equal(begin(reader1), end(reader1),
                                        begin(reader2) )
                          << std :: endl ;
```