Coroutines in C++17

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

CppNow 2015 • Gor Nishanov (gorn@microsoft.com) • Microsoft

What this talk is about?

- Stackless Resumable Functions
  - Lightweight, customizable coroutines
  - C++17 (maybe)
  - Experimental Implementation in MSVC 2015, Clang in progress, EDG

2012 - N3328
2013 - N3564
2013 - N3650
2013 - N3722
2014 - N3858
2014 - N3977
2014 - N4134 EWG direction approved
2014 - N4286
2015 - N4403 EWG accepted, sent to Core WG

http://isocpp.org/files/papers/N4402.pdf

http://isocpp.org/files/papers/N4403.pdf

Deon Brewis
Niklas Gustafsson
Herb Sutter
Jim Radigan
Daveed Vandevoorde

Coroutines are popular!

DART 1.9

```
Future<int> getPage(t) async {
 var c = new http.Client();
 try {
  var r = await c.get('http://url/search?q=$t');
  print(r);
  return r.length()
 } finally
  await
 }
}
```

Python: PEP 0492 (accepted on May 5, 2015)

```
ync def abinary(n):
  if n <= 0:
      return 1
  l = await abinary(n - 1)
  r = await abinary(n - 1)
  turn l + 1 + r
```

C#

```
async Task<string> WaitAsynchronouslyAsync()
{
    await   Task.Delay(10000);
    return "Finished";
}
```

C++17

```
future<string> WaitAsynchronouslyAsync()
{
    await   sleep_for(10ms);
    return "Finished"s;
}
```

HACK (programming language)

```
async function gen1(): Awaitable<int> {
 $x = await Batcher::fetch(1);
 $y = await Batcher::fetch(2);
 return $x + $y;
}
```

Design Goals

- **Scalable** (to hundred millions of concurrent coroutines)
- **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

2 x 2 x 2

- Two new keywords(*)
  - **await**
  - **yield**
- Two new concepts
  - Awaitable
  - Coroutine Promise
- Two new types
  - coroutine_handle
  - coroutine_traits

(*) may change based on discussion at the Lenexa last week

Coroutines

57 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

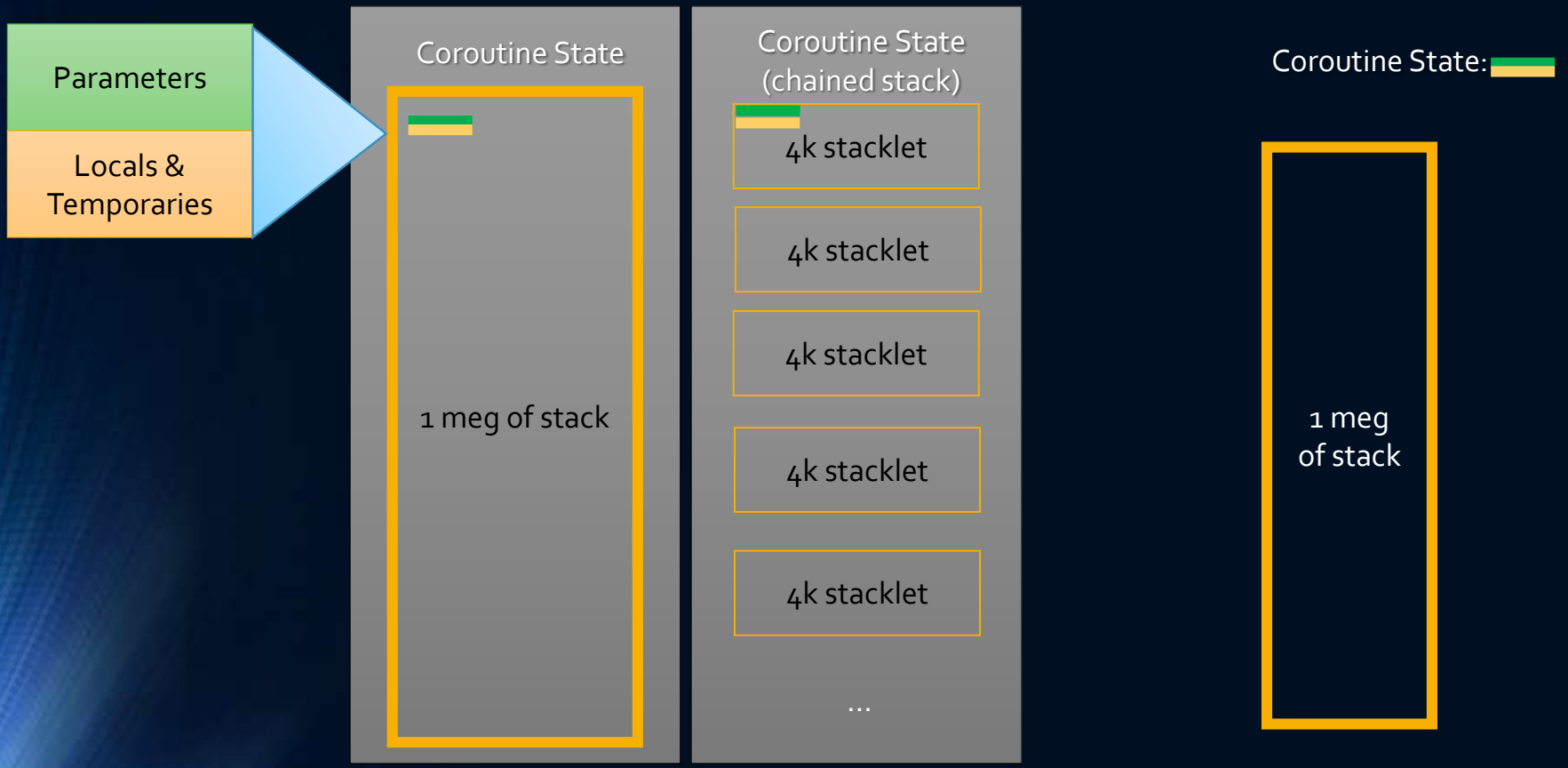
Coroutine classification

User Mode Threads / Fibers

Stackless Resumable Functions

- Symmetric / Asymmetric
  - Modula-2 / Win32 Fibers / Boost::context are symmetric (SwitchToFiber)
  - C#,Dart,Hack,etc. 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



Coroutines in C++

8.4.4 Coroutines **[dcl.fct.def.coroutine]**

1 A function is a *coroutine* if it contains one or more suspend-resume-points introduced by a potentially-evaluated `await` operator (5.3.8) expression and a `yield` statement (6.6.5).

2 [*Note:* From the perspective of the caller, a coroutine is just a function with that particular signature. The fact that a function is implemented as a coroutine is unobservable by the caller. —*end note*]

```
int main() {
   auto hello = [] {
     for (auto ch: "Hello, world\n")
        yield ch;
   };

   for (auto ch : hello()) cout << ch;
}
```

```
future<void> sleepy() {
   cout << "Going to sleep…\n";
   await sleep_for(1ms);
   cout << "Woke up\n";
   return 42;
}

int main() {
   cout << sleepy.get();
}
```

When would you want a coroutine?

Interleave

```
int main() {
  vector<int> a{ 1,2,3,4,5,6,7,8,9 };
  vector<int> b{ 10,20,30,40,50 };
  vector<int> c{ 100,200,300,400 };

  using T = decltype(c.begin());
  vector<Range_t<T>> rg{ Range(a), Range(b), Range(c) };

  for (auto v : interleave(rg)) {
     cout << v << ' ';
}
```

Output:

```
1 10 100 2 20 200 3 30 300 4 40 400 5 50 6 7 8 9
```

Not a coroutine (yet)

```
template <typename RangeOfRanges>
auto interleave(RangeOfRanges rg)

{
   using T = remove_reference_t<decltype(*begin(rg))>;
   vector<T> ranges(begin(rg), end(rg));

   for (;;) {
      int values_yielded_this_iteration = 0;
      for (auto && v : ranges) {
        if (begin(v) != end(v)) {
           cout << *begin(v++);
           ++values_yielded_this_iteration;
        }
      }
      if (values_yielded_this_iteration == 0)
        return;
    }
}
```

A generator coroutine !

```
template <typename RangeOfRanges>
auto interleave(RangeOfRanges rg)

{
   using T = remove_reference_t<decltype(*begin(rg))>;
   vector<T> ranges(begin(rg), end(rg));

   for (;;) {
      int values_yielded_this_iteration = 0;
      for (auto && v : ranges) {
        if (begin(v) != end(v)) {
           yield *begin(v++);
           ++values_yielded_this_iteration;
        }
      }
      if (values_yielded_this_iteration == 0)
        return;
    }
}
```

A generator coroutine !

```
template <typename RangeOfRanges>
auto interleave(RangeOfRanges rg)
   -> generator<whatever-is-being-yielded>
{
   using T = remove_reference_t<decltype(*begin(rg))>;
   vector<T> ranges(begin(rg), end(rg));

   for (;;) {
      int values_yielded_this_iteration = 0;
      for (auto && v : ranges) {
        if (begin(v) != end(v)) {
           yield *begin(v++);
           ++values_yielded_this_iteration;
        }
      }
      if (values_yielded_this_iteration == 0)
        return;
    }
}
```

When would you want a coroutine? Part II

Sync IO

```
auto tcp_reader(int total) -> ptrdiff_t
{
    ptrdiff_t result = 0;
    char buf[64 * 1024];
    auto conn = Tcp::ConnectSync("127.0.0.1", 1337);
    do
    {
        auto bytesRead = conn.readSync(buf, sizeof(buf));
        total -= bytesRead;
        result += std::count(buf, buf + bytesRead, 'c');
    }
    while (total > 0);
    return result;
}
```

Async IO

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

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}

STL looks like the machine language macro library of an anally retentive assembly language programmer

Pamela Seymour, Leiden University

Layered complexity

- Everybody
  - Safe by default, novice friendly

    Use coroutines and awaitables defined by standard library and boost and other high quality libraries
- Power Users
  - Define new awaitables to customize await for their environment
- Experts
  - Define new coroutine types

Compiler, Glue, Library

3 A coroutine needs a set of related types and functions to complete the definition of its semantics. These types and functions are provided as a set of member types or typedefs and member functions in the specializations of class template `std::experimental::coroutine_traits` (18.11.1).

Compiler

Types:

coroutine_traits

coroutine_handle

Concepts:

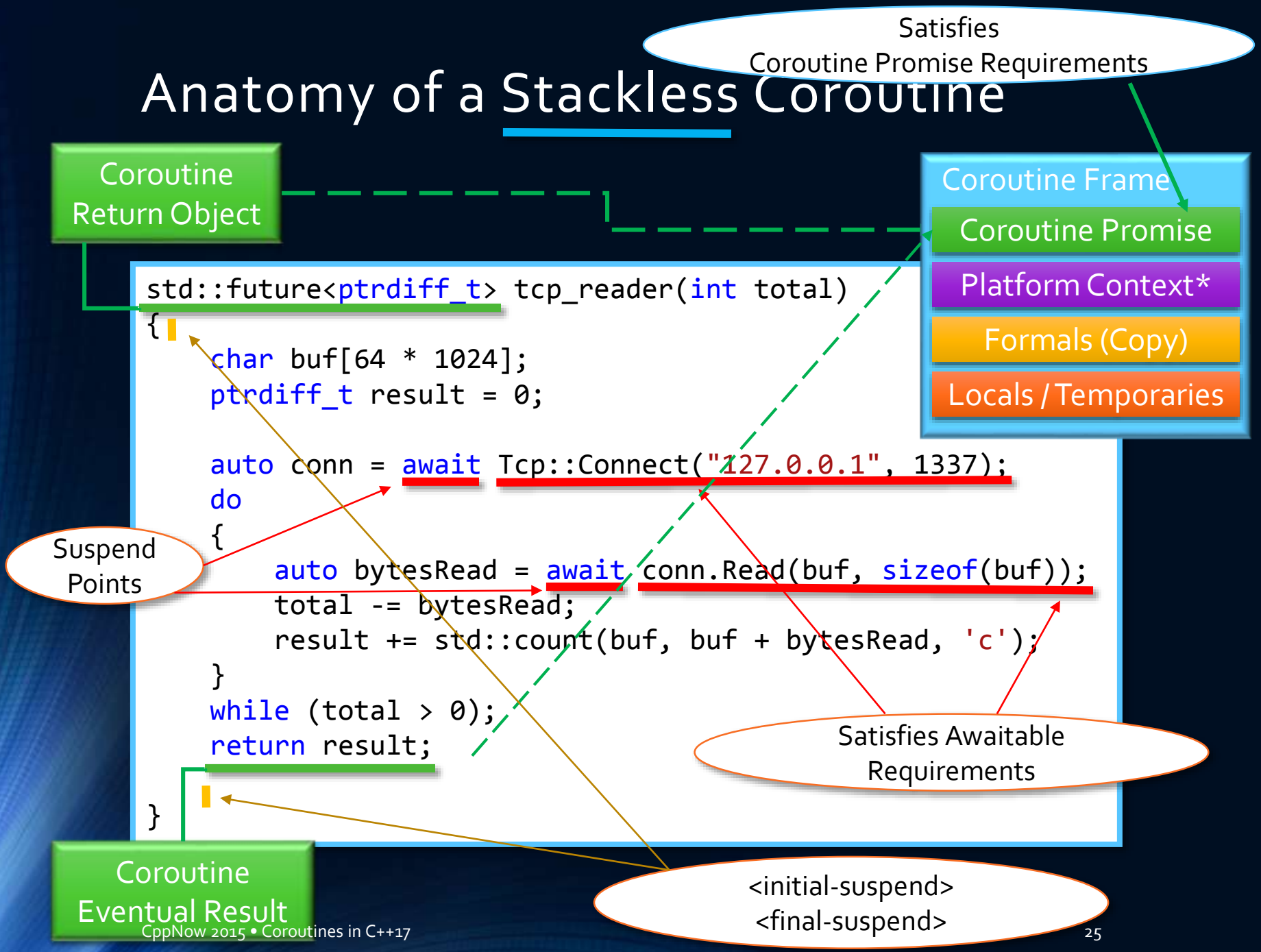
Awaitable

Coroutine Promise

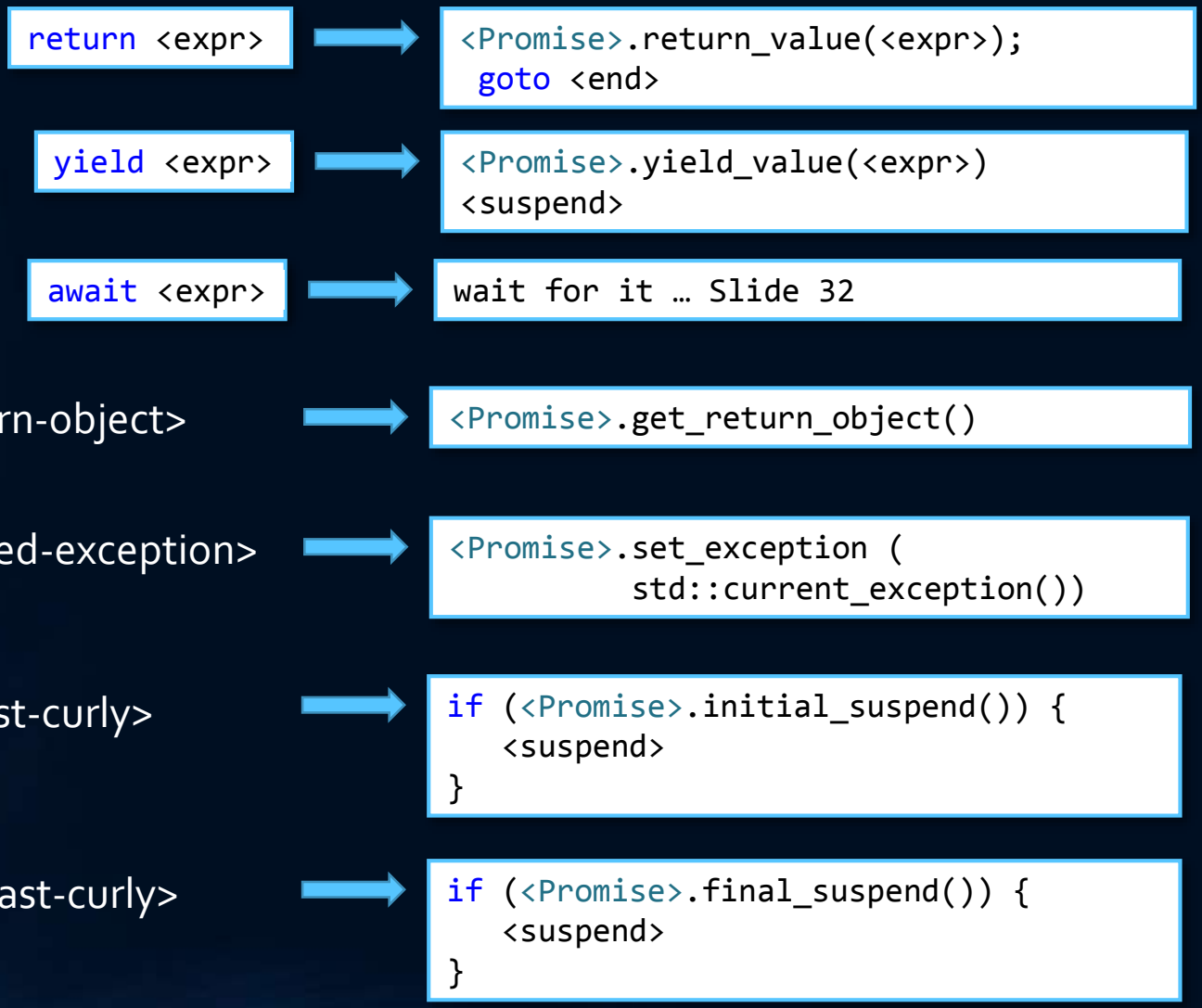
Libraries

generator<T>
async_generator<T>
future<T>
boost::future<T>
....

Anatomy of a Stackless Coroutine



Compiler vs Coroutine Promise



How does it work?

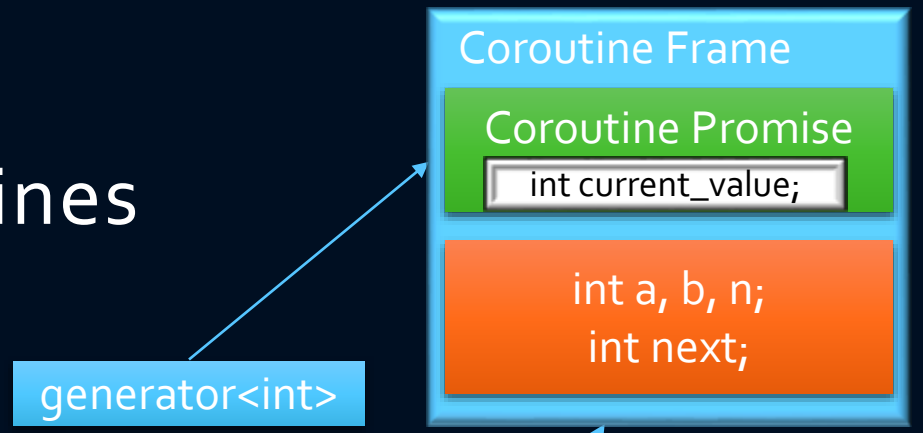
What is this?

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

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 << ' ';
        }
    }
}
```

Call

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

```
generator<int> __range; // raw
fib(&__range, 35)
```

RCX = &__range
RDX = 35

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

RAX = &__range

RSP

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

Heap

Coroutine Promise
saved RDI
saved RSI
saved RBX
saved RIP

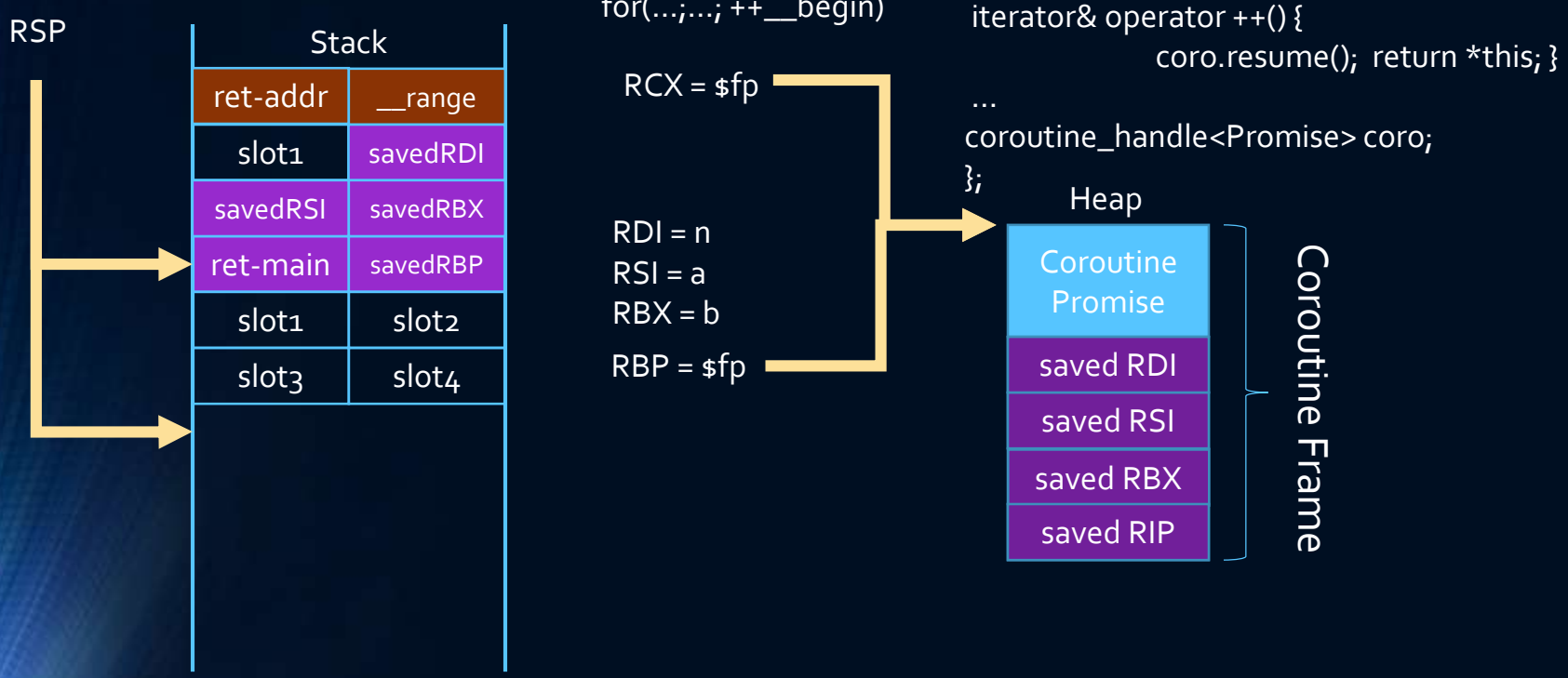
Coroutine Frame

Suspend!!!!

generator<int>

Resume

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



Awaitable

await <expr>

Expands into an 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(<coroutine-handle>);
  }
  return __tmp.await_resume();
}
```

suspend

resume

await <expr>

Expands into an expression equivalent of

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

```
{
  auto && __tmp = <expr>;
  if (! await_ready(__tmp)) {
      await_suspend(__tmp, <coroutine-handle>);
                                                    suspend
                                                    resume
  }
  return await_resume(__tmp);
}
```

Trivial Awaitable #1

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

Trivial Awaitable #1

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

```
await suspend_always {};
```

Trivial Awaitable #2

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

Simple Awaitable #1

```
std::future<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);}
};
```

Do not use!
For illustration only!

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
  - coroutine_handle
  - coroutine_traits

coroutine_handle

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

template <> struct coroutine_handle<void> {
   void resume();
   void destroy();
   bool done() const;
   void * to_address();
   static coroutine_handle<void> from_address(void*);
   void operator()(); // same as resume()
…
};
```

== != < > <= >=

Simple Awaitable #2: Raw OS APIs

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

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

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

    void await_suspend(std::coroutine_handle<> h)  {
       int64_t relative_count = -duration.count();
       timer = CreateThreadpoolTimer(TimerCallback, h.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
  - coroutine_handle
  - coroutine_traits

coroutine_traits

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

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

```
template <typename R, typename... Ts>
struct coroutine_traits {
   using promise_type = typename R::promise_type;
};
```

Defining Coroutine Promise for boost::future

```
namespace std {
  template <typename T, typename… anything>
  struct coroutine_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 return_value(U && value) {
            promise.set_value(std::forward<U>(value));
        }

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

Awaitable and Exceptions

coroutine_handle

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

template <> struct coroutine_handle<void> {
   void resume();
   void destroy();
   bool done() const;
   void * to_address();
   static coroutine_handle<void> from_address(void*);
   void operator()(); // same as resume()
…
};

template <typename Promise>
struct coroutine_handle: public coroutine_handle<> {
   Promise & promise();
   explicit coroutine_handle(Promise*);
   …
};
```

== != < > <= >=

Exceptionless Error Propagation (Part 1/3)

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

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

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

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

Exceptionless Error Propagation (Part 2/3)

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

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

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

   template <class T, class Promise>
   void await_suspend(
      unique_future<T> & t, std::coroutine_handle<Promise> h)
   {
       t.then([=](auto& result){
          if(result.has_exception()) {
            h.promise().set_exception(result.get_exception_ptr());
            h.destroy();
          }
          else
             h.resume();
       });
   }
}
```

Exceptionless Error Propagation (Part 3/3)

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

namespace boost {
   template <class T>
   bool await_ready(unique_future<T> & t) { return t.is_ready() &&
                                                   !t.has_exception();}
   template <class T>
   auto await_resume(unique_future<T> & t) { return t.get(); }

   template <class T, class Promise>
   void await_suspend(
      unique_future<T> & t, std::coroutine_handle<Promise> h)
   {
       t.then([=](auto& result){
          if(result.has_exception()) {
            h.promise().set_exception(result.get_exception_ptr());
            h.destroy();
          }
          else
             h.resume();
       });
   }
}
```

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;

}
```

Expected<T>, yeah!

```
expected<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;

}
```

Beyond Coroutines

```
M<T> f()
{
    auto x = await f1();
    auto y = await f2();
    return g(x,y);
}
```

Where
- f1: () → M'<X>
- f2: () → M''<Y>
- g: (X,Y) → T
- await: M*<T> → T
- return: T → M<T>

await: unwraps a value from a container M*<T>
return: puts a value back into a container M<T>

Future<T>: container of T, unwrapping strips temporal aspect
optional<T>: container of T, unwrapping strips "not there aspect"
expected<T>: container of T, unwrapping strips "or an error aspect"
std::future<T>: unwrapping strips temporal and may have error aspects

Beyond Coroutines: Constexpr Generators

```
constexpr auto strided_init(
  int from, int to, int step)
{
    while (from < to) {
        yield from;
        from += step;
    }
}

int a[] = {strided_init(10,100,5)};
```

Keywords

A

```
future<int> Sum(async_stream<int> & input)
{
    int sum = 0;
    for await(auto v: input)
        sum += v;
    return sum;
}
```

```
future<int> sleepy() {
   cout << “Going to sleep…\n";
   await sleep_for(1ms);
   cout << “Woke up\n";
   return 42;
}
```

```
auto flatten(node* n) {
    if (n == nullptr) return;
    yield flatten(n->left);
    yield n->value;
    yield flatten(n->right);
}
```

B

```
future<int> Sum(async_stream<int> & input)
{
    int sum = 0;
    co_for(auto v: input)
        sum += v;
    co_return sum;
}
```

```
future<int> sleepy() {
   cout << “Going to sleep…\n";
   co_await sleep_for(1ms);
   cout << “Woke up\n";
   co_return 42;
}
```

```
auto flatten(node* n) {
    if (n == nullptr) co_return;
    co_yield flatten(n->left);
    co_yield n->value;
    co_yield flatten(n->right);
}
```

C

```
future<int> Sum(async_stream<int> & input)
{
    int sum = 0;
    for coawait(auto v: input)
        sum += v;
    coreturn sum;
}
```

```
future<int> sleepy() {
   cout << “Going to sleep…\n";
   coawait sleep_for(1ms);
   cout << “Woke up\n";
   coreturn 42;
}
```

```
auto flatten(node* n) {
    if (n == nullptr) coreturn;
    coyield flatten(n->left);
    coyield n->value;
    coyield flatten(n->right);
}
```

The End

C++ will stand out even more!

DART

```
Future getPage(t) async {
 var c = new http.Client();
 try {
  var r = await c.get('http://url/search?q=$t');
  print(r);
 } finally {
  await
 }
}
```

Python: PEP 0492 (accepted on May 5, 2015)

```
ync def abinary(n):
  if n <= 0:
      return 1
  l = await abinary(n - 1)
  r = await abinary(n - 1)
  urn l + 1 + r
```

C#

```
async Task<string> WaitAsynchronouslyAsync()
{
    await   Task.Delay(10000);
    return "Finished";
}
```

C++17

```
auto WaitAsynchronouslyAsync()
{
    co_await   sleep_for(10ms);
    co_return "Finished";
}
```

HACK (programming language)

```
async function gen1(): Awaitable<int> {
 $x = await Batcher::fetch(1);
 $y = await Batcher::fetch(2);
 return $x + $y;
}
```

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>
  - …

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;
   }
}
```

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;
   }
}
```