accu 2022

POWERFUL DATA PROCESSING PIPELINES WITH C++20 COROUTINES

ANDREAS WEIS

Powerful Data Processing Pipelines

With C++20 Coroutines

Andreas Weis

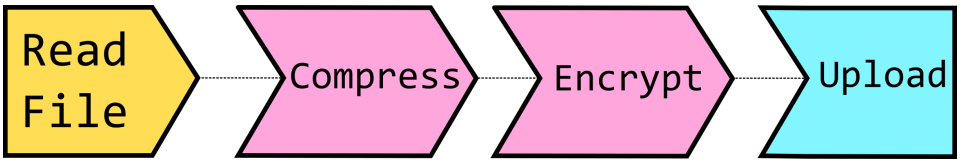
Woven Planet

ACCU 2022

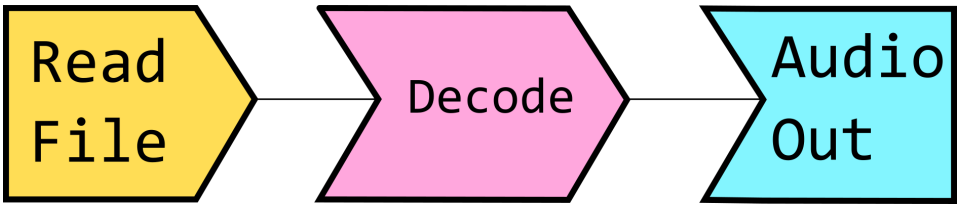
About me - Andreas Weis (he/him)

- ComicSansMS
- @DerGhulbus
- Co-organizer of the Munich C++ User Group
- Currently working as a Runtime Engineer for Woven Planet

Motivation - File backup



Motivation - Media streaming



Motivation - Design Goals

- Be able to implement arbitrary processing steps
- Have a unified way to compose steps into a full pipeline
- Detailed control about the control flow (i.e. when and how does each step get executed)
- Decouple those concerns as much as possible in the implementation, so that they can be changed independently

Motivation - Design Goals

```
FileSource{ from_filename }
  | Compress{}
  | Encrypt{ private_key }
  | NetworkSink{ to_url };
```

Overview

- A general interface for data processing steps
- Composing several processing steps into a pipeline
- Pipeline control flow
- General introduction to coroutines
- Coroutines and pipeline control flow

Warning! Dangerous slide code ahead!

- The code examples in this presentation are meant to illustrate ideas
- They are in no way fit for production and often intentionally omit details like `const` or `noexcept` for brevity
- Error handling is also frequently missing in code examples
- For coroutines I will simplify things and omit details. Check cppreference when implementing your own to understand all the details.
- Exercise caution when reimplementing this for production

Anatomy of a pipeline

- 

Source
- 

Sink
- 

Filter

A universal interface for processing steps

```
zstream zs;
// [...]
zs.next_in = in_buffer;
zs.avail_in = in_buffer_size;
zs.next_out = out_buffer;
zs.avail_out = out_buffer_size;
auto const result = deflate(&zs, Z_NO_FLUSH);
assert(result == Z_OK);
assert((zs.avail_in == 0) || (zs.avail_out == 0));
```

The zlib interface

- Before each call to deflate, input and output buffers need to be set
- A call will process data until either the input is depleted or the output is filled up
- Keep calling deflate() in a loop, refilling buffers as needed, until all input has been processed
- After all input has been processed, processing enters a finalization phase where all remaining output data is flushed

C++ filter interface

```
enum class ProcessReturn { Pending, Done, Error };

struct Filter {
  std::span<std::byte const> in;
  std::span<std::byte> out;
  ProcessReturn process();
};
```

C++ filter interface

```
template<typename T>
concept PipelineStage = requires(T a) {
    { a.process() } -> std::same_as<ProcessReturn>;
};

template<typename T>
concept PipelineSource = PipelineStage<T> &&
  requires(T a) {
    { a.out } -> std::same_as<std::span<std::byte>&>;
};
```

1

[1] The & in same_as is needed because of https://stackoverflow.com/a/62792349/577603

C++ filter interface

```
template<typename T>
concept PipelineSink = PipelineStage<T> &&
  requires(T a) {
    { a.in } ->
       std::same_as<std::span<std::byte const>&>;
};

template<typename T>
concept PipelineFilter =
  PipelineSource<T> && PipelineSink<T>;
```

Example: File Source


```
struct FileSource { FILE* fin; /* ... */ };
FileSource::FileSource(std::string_view fname);

ProcessReturn FileSource::process() {
  if (!fin) { return ProcessReturn::Error; }
  if (out.empty()) { return ProcessReturn::Error; }

  std::size_t const bytes_read =
    fread(out.data(), 1, out.size(), fin);
  out = out.subspan(bytes_read);

  // ...
```

Example: File Source


```
    // ...
    if (feof(fin)) {
        fclose(fin);
        fin = nullptr;
        return ProcessReturn::Done;
    }

    if (ferror(fin)) {
        return ProcessReturn::Error;
    }

    return ProcessReturn::Pending;
}
```

Example: Identity Filter


```
ProcessReturn FilterId::process() {
  if (in.empty()) { return ProcessReturn::Done; }
  if (out.empty()) { return ProcessReturn::Error; }

  std::size_t const bytes_to_copy =
    std::min(in.size(), out.size());
  std::memcpy(out.data(), in.data(), bytes_to_copy);
  in = in.subspan(bytes_to_copy);
  out = out.subspan(bytes_to_copy);

  return ProcessReturn::Pending;
}
```

Example: Deflate Filter


```
#include <zlib.h>
struct FilterDeflate {
  // ...
  z_stream zs;
};
FilterDeflate::FilterDeflate() {
  std::memset(&pimpl->zs, 0, sizeof(z_stream));
  deflateInit(&pimpl->zs, Z_BEST_COMPRESSION);
}
FilterDeflate::~FilterDeflate() {
  deflateEnd(&pimpl->zs);
}
```

Example: Deflate Filter


```
ProcessReturn FilterDeflate::process() {
  bool do_flush = false;
  if (zs.avail_in == 0) {
    if (in.empty()) {
      do_flush = true;
    } else {
      zs.next_in =
        reinterpret_cast<Bytef const*>(in.data());
      zs.avail_in = static_cast<uInt>(in.size());
      do_flush = false;
    }
  }
  // ...
```

Example: Deflate Filter


```
// ...
if (zs.avail_out == 0) {
  if (out.empty()) { return ProcessReturn::Error; }
  zs.next_out = reinterpret_cast<Bytef*>(out.data());
  zs.avail_out = static_cast<uInt>(out.size());
}

// ...
```

Example: Deflate Filter


```
// ...
if (!do_flush) {
  auto const res = deflate(&zs, Z_NO_FLUSH);
  if (res != Z_OK) { return ProcessReturn::Error; }
  std::size_t in_consumed = in.size() - zs.avail_in;
  in = in.subspan(in_consumed);
  std::size_t out_consumed = out.size() - zs.avail_out;
  out = out.subspan(out_consumed);
// ...
```

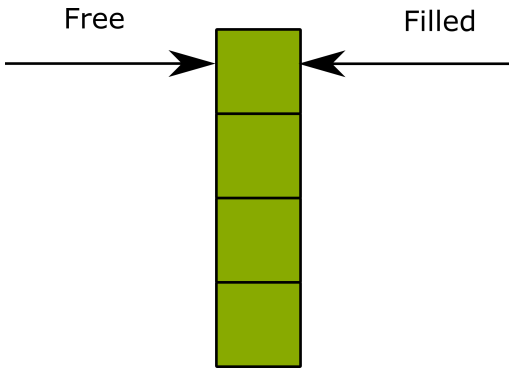
Example: Deflate Filter


```
    // ...
    } else /* do_flush */ {
      auto const res = deflate(&zs, Z_FINISH);
      if ((res != Z_OK) && (res != Z_STREAM_END))
      { return ProcessReturn::Error; }
      out = out.subspan(out.size() - zs.avail_out);
      if (res == Z_STREAM_END) {
        deflateReset(&zs);
        return ProcessReturn::Done;
      }
    }
    return ProcessReturn::Pending;
}  // process()
```

Connecting multiple stages through buffers

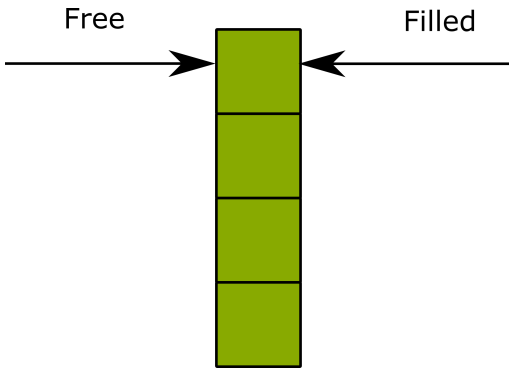
```
class RingBuffer {
public:
  RingBuffer(size_t n_buffers, size_t buffer_size);
  AcquiredBuffer acquireFreeBuffer();
  AcquiredBuffer acquireFilledBuffer();
};
```

Ring Buffer



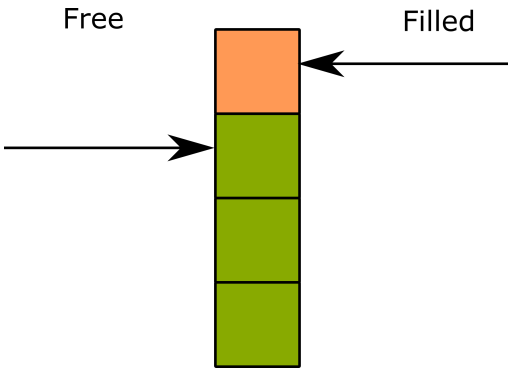

```
RingBuffer b{ 4, 1024 };
```

Ring Buffer



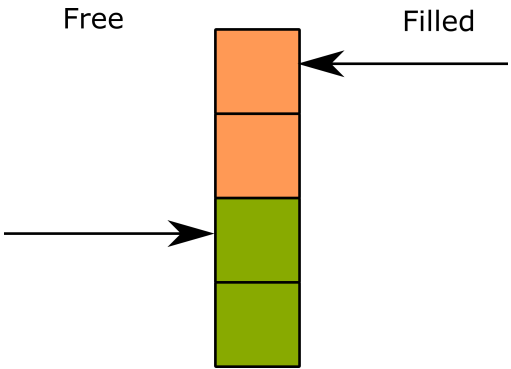

```
RingBuffer b{ 4, 1024 };
b.acquireFreeBuffer();
```

Ring Buffer



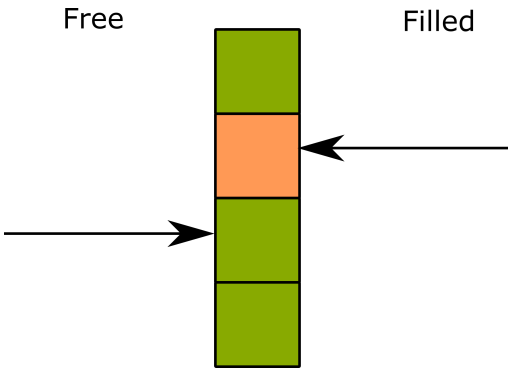

```
RingBuffer b{ 4, 1024 };
b.acquireFreeBuffer();
```

Ring Buffer



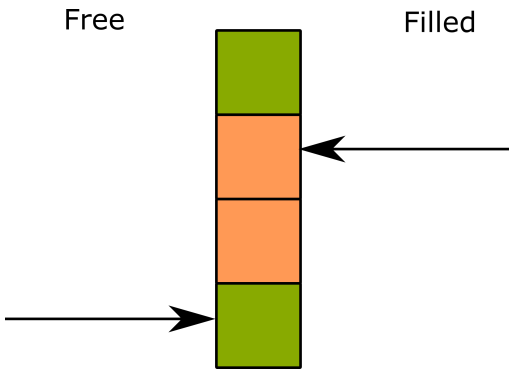

```
RingBuffer b{ 4, 1024 };
b.acquireFreeBuffer();
b.acquireFreeBuffer();
```

Ring Buffer



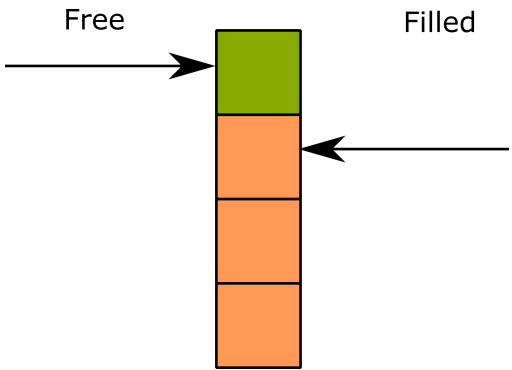

```
RingBuffer b{ 4, 1024 };
b.acquireFreeBuffer();
b.acquireFreeBuffer();
b.acquireFilledBuffer();
```

Ring Buffer



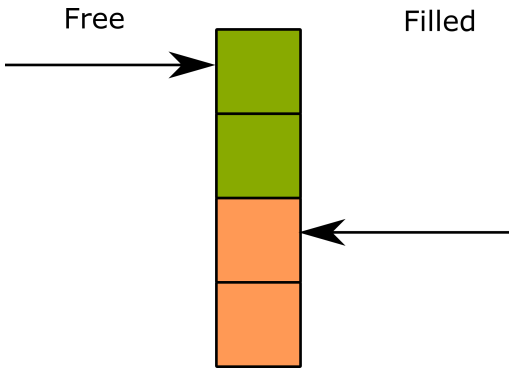

```
RingBuffer b{ 4, 1024 };
b.acquireFreeBuffer();
b.acquireFreeBuffer();
b.acquireFilledBuffer();
b.acquireFreeBuffer();
```

Ring Buffer



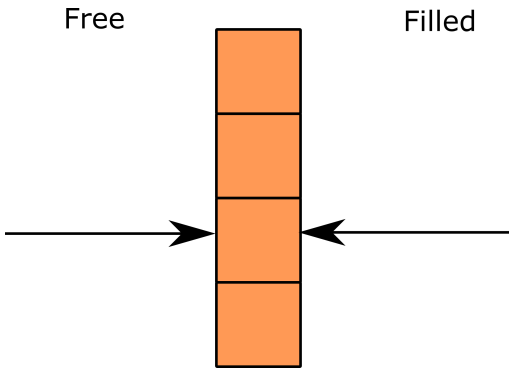

```
RingBuffer b{ 4, 1024 };
b.acquireFreeBuffer();
b.acquireFreeBuffer();
b.acquireFilledBuffer();
b.acquireFreeBuffer();
b.acquireFreeBuffer();
```

Ring Buffer




```
RingBuffer b{ 4, 1024 };
b.acquireFreeBuffer();
b.acquireFreeBuffer();
b.acquireFilledBuffer();
b.acquireFreeBuffer();
b.acquireFreeBuffer();
b.acquireFilledBuffer();
```

Ring Buffer

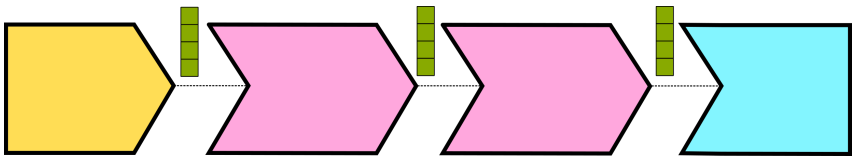



```
RingBuffer b{ 4, 1024 };
b.acquireFreeBuffer();
b.acquireFreeBuffer();
b.acquireFilledBuffer();
b.acquireFreeBuffer();
b.acquireFreeBuffer();
b.acquireFilledBuffer();
b.acquireFreeBuffer();
b.acquireFreeBuffer();
```

Acquired Buffer Interface

```
class AcquiredBuffer {
public:
  AcquiredBuffer(AcquiredBuffer&& rhs) noexcept;
  ~AcquiredBuffer();
  void release();
  void commitBytes(std::size_t n);
  std::span<std::byte> getBuffer() & noexcept;
};
{
  AcquiredBuffer acq = b.acquireFreeBuffer();
  size_t const bytes_written = writeTo(acq.getBuffer());
  acq.commitBytes(bytes_written);
}
```

Connecting steps with buffers



- A buffer is inserted between any two stages
- The buffer is used as output by the stage to its left and as input by the stage to its right.
- Any communication from one stage to its neighbours happens through the buffer.

Connecting multiple stages through buffers

```
enum class OpState {
  Run, Finalize, Done, Error, Abort
};


class RingBuffer {
public:
  RingBuffer(size_t n_buffers, size_t buffer_size);
  AcquiredBuffer acquireFreeBuffer();
  AcquiredBuffer acquireFilledBuffer();

  OpState operation_state;
};
```

Pipeline Control Flow

Recap of what we have so far:

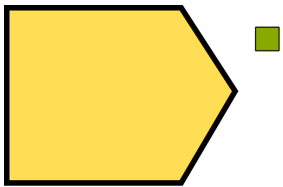
- Source, Sink and Filter

```
struct Filter {
  std::span<std::byte> in;
  std::span<std::byte> out;
  ProcessReturn process();
};
```

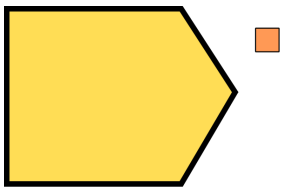
- RingBuffer

```
struct RingBuffer {
  AcquiredBuffer acquireFreeBuffer();
  AcquiredBuffer acquireFilledBuffer();
  OpState operation_state;
};
```

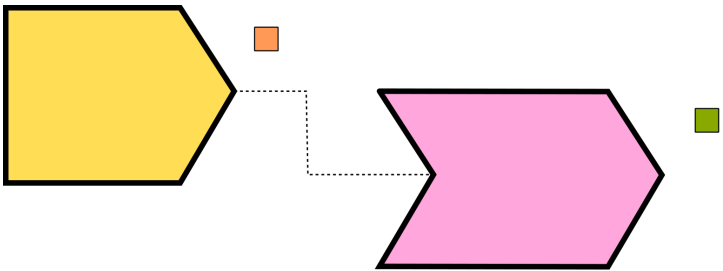
Synchronous Case: Producer-driven



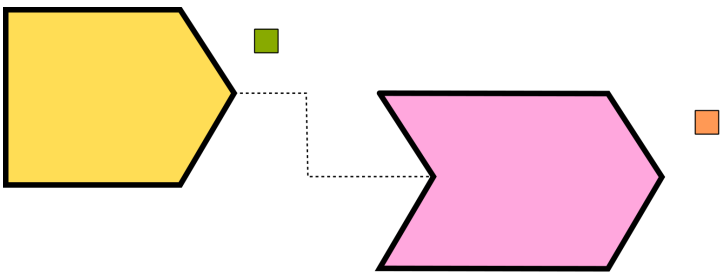
Synchronous Case: Producer-driven



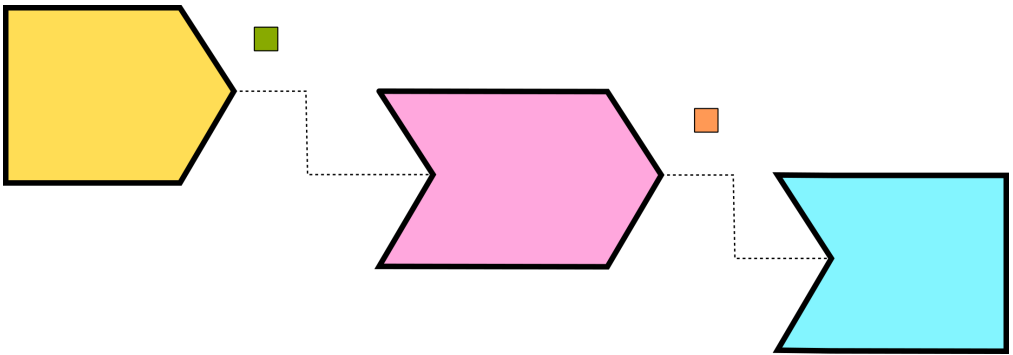
Synchronous Case: Producer-driven



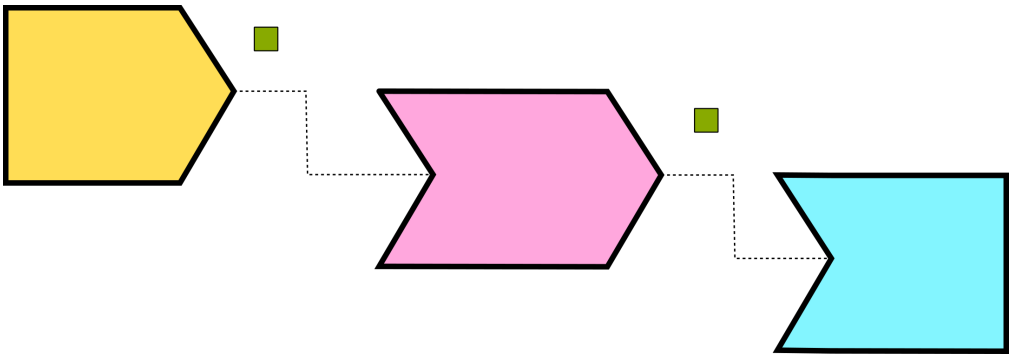
Synchronous Case: Producer-driven



Synchronous Case: Producer-driven



Synchronous Case: Producer-driven



Synchronous Case: Producer-driven


```
void executeStep(PipelineSource auto& source,
                 RingBuffer& b_out, auto&& downstream) {
  while (true) {
    auto acq = b_out.acquireFreeBuffer();
    source.out = acq.getBuffer();
    ProcessReturn ret = source.process();
    if (ret == ProcessReturn::Done) { break; }
    size_t const bytes_written =
      acq.getBuffer().size() - source.out.size();
    acq.commitBytes(bytes_written);
    downstream();
  }
}
```

Synchronous Case: Producer-driven


```
void executeStep(PipelineSink auto& sink,
                 RingBuffer& b_in) {
  auto acq = b_in.acquireFilledBuffer();
  source.in = acq.getBuffer();
  ProcessReturn ret = source.process();
  if (ret == ProcessReturn::Done) { return; }
}


executeStep(source, buffer, [&]() {
    executeStep(sink, buffer);
  });
```

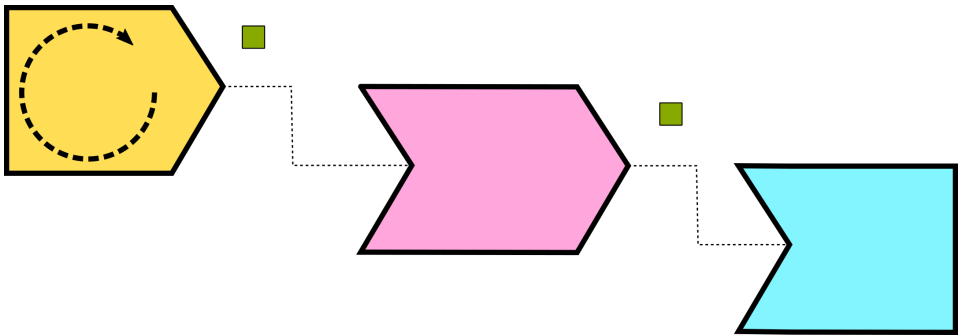
Synchronous Case: Producer-driven

Just combine source and sink steps?

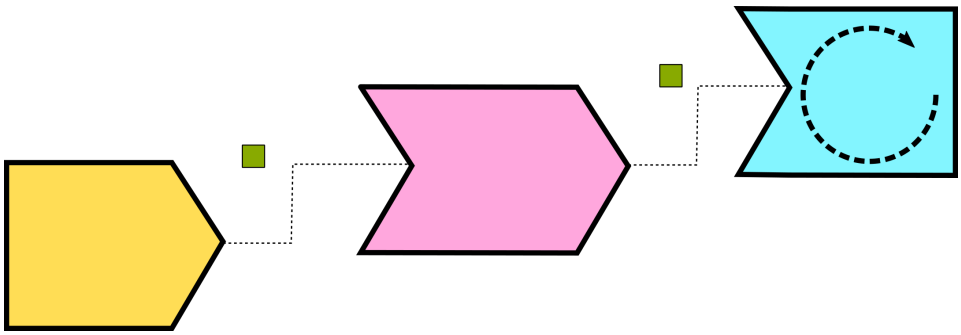
```
void executeStep(PipelineFilter auto& filter,
                 RingBuffer& b_in,
                 RingBuffer& b_out,
                 auto&& downstream)
```

- Processing may empty either the in buffer, or the out buffer, but not necessarily both
- If the out buffer is not filled, we need to return back to the producer to get another input buffer filled
- But what do we do with the `AcquiredBuffer` for the output in that case? RAII no longer working nicely.

Synchronous Case: Producer-driven vs. Consumer-driven



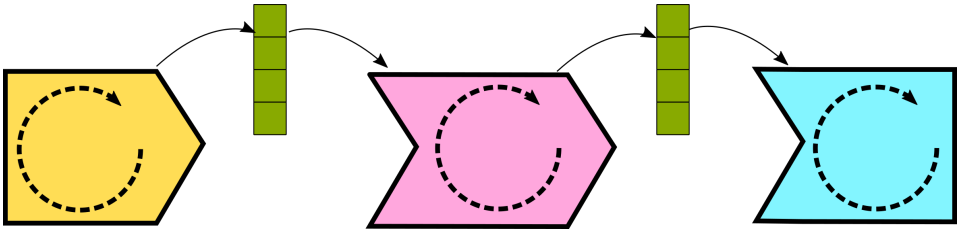
Synchronous Case: Producer-driven vs. Consumer-driven



Synchronous Case: Producer-driven vs. Consumer-driven

```
// producer-driven:
executeStep(source, buffer, [&]() {
    executeStep(sink, buffer);
  });

// consumer-driven:
executeStep(sink, buffer, [&]() {
    executeStep(source, buffer);
  });
```

Parallel case



Ring buffer is now accessed concurrently!

Thread safe Ring Buffer access

Single producer, single consumer Ring Buffer

```
struct RingBuffer {
  AcquiredBuffer acquireFreeBuffer();
  AcquiredBuffer acquireFilledBuffer();
  OpState operation_state;
};
```

Thread safe Ring Buffer access

Single producer, single consumer Ring Buffer

```
struct MTRingBuffer {
  MTAcquiredBuffer acquireFreeBuffer();
  MTAcquiredBuffer acquireFilledBuffer();
  RingBuffer b_;
};
```

Thread safe Ring Buffer access

```
MTAcquiredBuffer MTRingBuffer::acquireFilledBuffer()
{
  std::unique_lock lk{ mtx };
  AcquiredBuffer ret;
  cv.wait(lk, [this, &ret]() -> bool {
    ret = b_.acquireFilledBuffer();
    return static_cast<bool>(ret) ||
      (b_.operation_state() != OpState::Run);
  });
  return MTAcquiredBuffer{ this, std::move(ret) };
}
```

Parallel Case

Source

```
ProcessReturn executeStep(PipelineSource auto& step,
                          PipelineBuffer auto& out_buffer) {
  ProcessReturn ret = ProcessReturn::Pending;
  while(ret == ProcessReturn::Pending) {
    auto acq = out_buffer.acquireFreeBuffer();
    step.out = acquired_buffer.getBuffer();
    ret = step.process();
    if (ret != ProcessReturn::Pending) { break; }
    acq.commitBytes(...);
  }
  return ret;
}
```

Parallel Case


```
ProcessReturn executeStep(PipelineSink auto& step,
                  PipelineBuffer auto& in_buffer) {
  ProcessReturn ret = ProcessReturn::Pending;
  while (ret == ProcessReturn::Pending) {
    auto acquired_buffer =
      in_buffer.acquireFilledBuffer();
    step.in = acquired_buffer.getBuffer();
    ret = step.process();
    if (ret != ProcessReturn::Pending) { break; }
  }
  return ret;
}
```

Parallel Case


```
ProcessReturn executeStep(PipelineFilter auto& step,
                          PipelineBuffer auto& in_buffer,
                          PipelineBuffer auto& out_buffer) {
  ProcessReturn ret = ProcessReturn::Pending;
  while (ret == ProcessReturn::Pending) {
    if (step.in.empty()) { /* acquire in */ }
    if (step.out.empty()) { /* acquire out */ }
    ret = step.process();
    // ...
  }
  return ProcessReturn::Error;
}
```

Parallel Case


```
template<PipelineBuffer InBuffer_T,
         PipelineBuffer OutBuffer_T>
ProcessReturn executeStep(PipelineFilter auto& step,
                          InBuffer_T& in_buffer,
                          OutBuffer_T& out_buffer)
{
    std::optional<typename InBuffer_T::BufferType>
      opt_acquired_in;
    std::optional<typename OutBuffer_T::BufferType>
      opt_acquired_out;
    // ...
}
```

Parallel Case


```
while (ret == ProcessReturn::Pending) {
  // acquire in
  if (step.in.empty()) {
    assert(!opt_acquired_in);
    opt_acquired_in = in_buffer.acquireFilledBuffer();
    step.in = opt_acquired_in->getBuffer();
  }
  // ...
  ret = step.process();
  if (step.in.empty()) {
    opt_acquired_in = std::nullopt;
  }
}
```

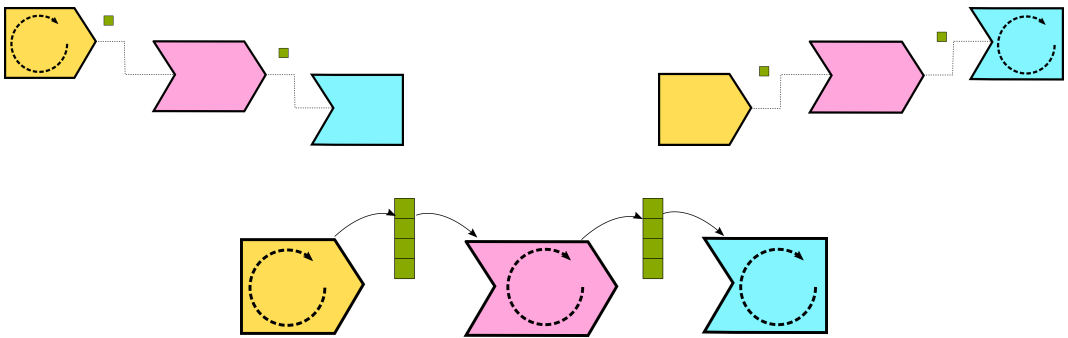
Parallel Case


```
while (ret == ProcessReturn::Pending) {
  // ...
  // acquire out
  if (step.out.empty()) {
    opt_acquired_out = out_buffer.acquireFreeBuffer();;
    step.out = opt_acquired_out->getBuffer();
  }
  ret = step.process();
  // ...
  if (step.out.empty()) {
    opt_acquired_out->commitBytes(size);
    opt_acquired_out = std::nullopt;
  }
```

Parallel Case - User View

```
FileSource source{ fname_src };
FileSink sink{ fname_dst };
FilterDeflate deflate;
MTRingBuffer b1, b2;
std::jthread t1{ [&] { executeStep(source, b1); } };
std::jthread t2{ [&] { executeStep(deflate, b1, b2); } };
std::jthread t3{ [&] { executeStep(sink, b2); } };
```

Recap: Control Flow



Coroutine Basics

- Coroutines are like a function that can be paused in the middle
- Execution can be suspended and resumed later with all surrounding function state still intact
- C++ provides stackless coroutines - suspension only affects the currently executing function, but not its parents

Coroutine Basics - User View

A coroutine is a function containing either

- `co_await`
- `co_return`
- `co_yield`

Coroutine Basics - Asynchronous computation

```
auto [ec, bytes_read] = read(socket, buffer);
// ...

async_read(socket, buffer,
  [](std::error_code ec, std::size_t bytes_read) {
    // ...
  });
```

Coroutine Basics - Asynchronous computation

```
auto [ec, bytes_read] = read(socket, buffer);
// ...

auto [ec, bytes_read] =
          co_await async_read(socket, buffer);
// ...
```

Coroutine Basics - Suspend/resume

```
MyCoroutine co = startComputation(initial_data);
auto some_results = co.provide(some_data);
auto more_results = co.provide(more_data);
auto final_results = co.results;
```

Participants in coroutine execution

- Return object
- Promise
- Awaitable

Return object

```
MyCoroutine co = startComputation(initial_data);
```

- Return type of the coroutine function
- Typically receives the `coroutine_handle` of the started coroutine as constructor argument
- Points out the promise through the nested `promise_type` typedef

Promise Type

- Main point of interaction from within the coroutine
- Constructed by the compiler at the start of the coroutine. Receives initial function arguments as constructor arguments
- Determines what happens at essential points in a coroutines' lifetime: Start and completion of execution, exit via `co_return` or exception
- Responsible for construction of return object through `get_return_object` (needs to be compatible but not same as return object type)
- Provides access to `coroutine_handle` via `coroutine_handle<Promise_T>::from_promise()`
- Also, promise can be retrieved from the handle via `promise()` member function on `coroutine_handle`

Awaitable

```
AsyncRead awaitable = async_read(socket, buffer);
auto [ec, bytes_read] = co_await awaitable;
```

- A type that can be co_awaited on.
- Provides hooks to inject code into the suspend and resume procedures
- May decide not to suspend at all
- Gets passed a handle to the running coroutine before suspension.

Minimum boilerplate

```
struct Coroutine {
  struct promise_type { /* ... */ };
};

Coroutine f1() {
  co_return;
}

int main() {
  Coroutine c1 = f1();
}
```

Minimum boilerplate

```
struct promise_type {
  Coroutine get_return_object();
  std::suspend_never initial_suspend();
  std::suspend_always final_suspend();
  void return_void();
  void unhandled_exception();
};
```

Minimum boilerplate

```
Coroutine f1(T... args) {
  // using promise = Coroutine::promise_type;
  // promise __p(args...);
  // __p.get_return_object(); -> to caller
  // co_await __p.initial_suspend()
}

int main() {
  Coroutine c1 = f1();
}
```

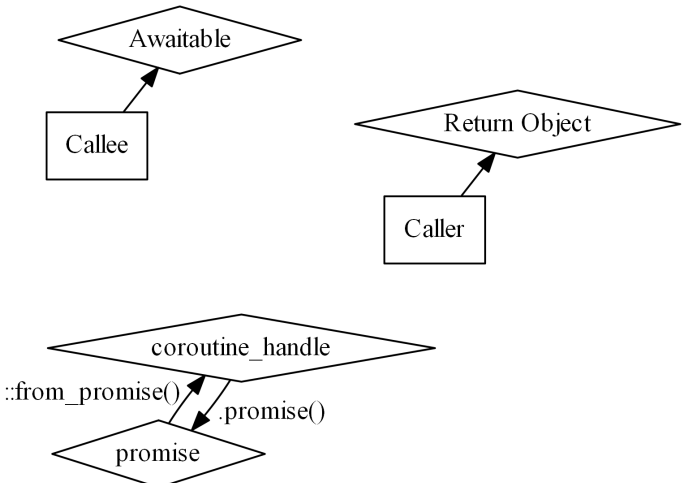
Minimum boilerplate

```
Coroutine f1(T... args) {
  co_return;
  // __p.return_void();
  // co_await __p.final_suspend()
  // __p.~promise();
}

int main() {
  Coroutine c1 = f1();
}
```

Suspension

```
struct MyAwaitable {
  bool await_ready() { return false; }
  void await_suspend(std::coroutine_handle<> h) {}
  void await_resume() {}
};

Coroutine f1(T... args) {
  co_await MyAwaitable{};
  fmt::print("Never reached\n");
}
```

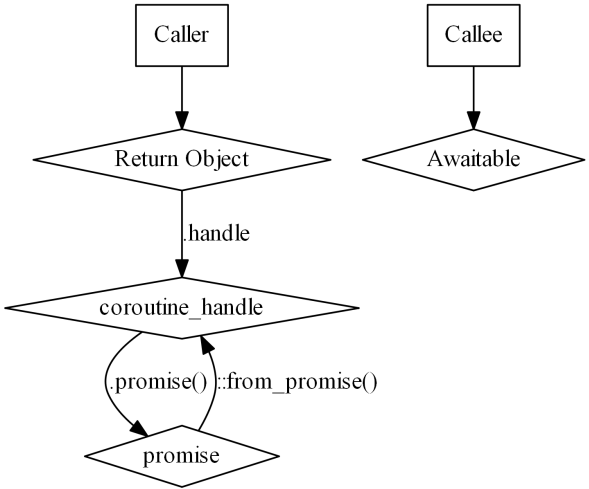
Making acquaintances



Making acquaintances

```
struct Coroutine {
  std::coroutine_handle<promise_type> handle;
  Coroutine(std::coroutine_handle<promise_type> h)
  : handle(h) {}
};
struct promise_type {
  coroutine get_return_object() {
    return coroutine{
      std::coroutine_handle<promise>::from_promise(*this)
      };
  }
};
```

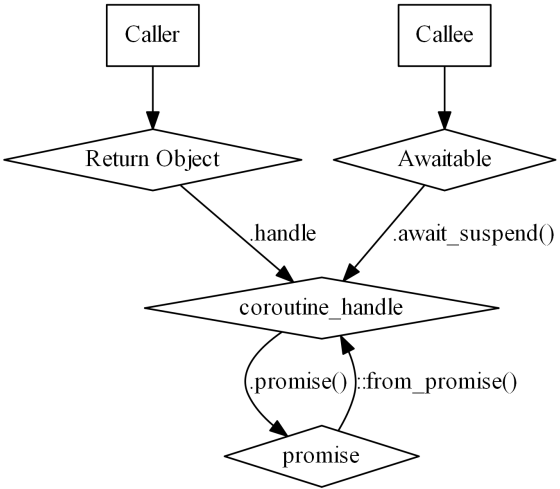
Making acquaintances



Making acquaintances

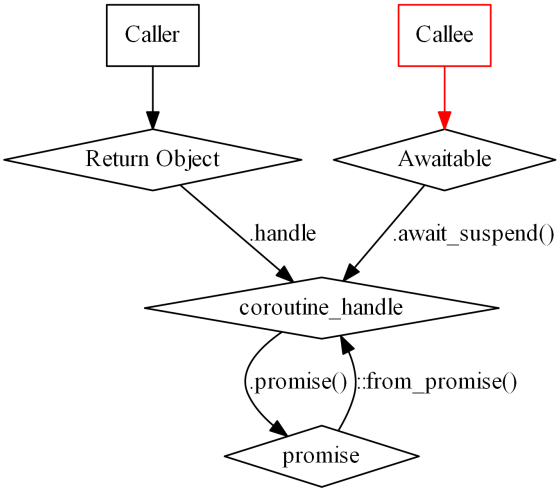
```
struct MyAwaitable {
  void await_suspend(std::coroutine_handle<> h) {}
};
```

Our (Incomplete) Map of Coroutine Land



Getting data out of a coroutine

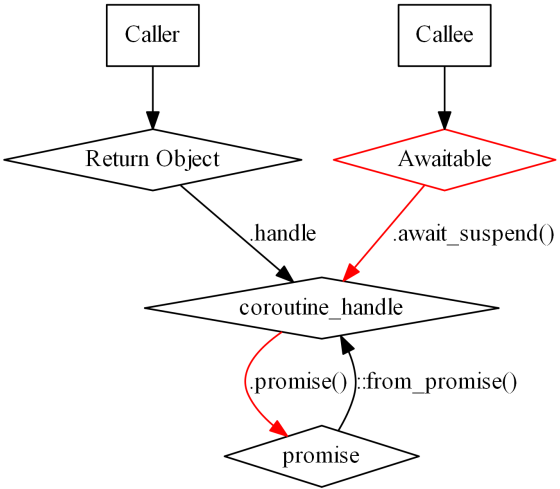
Getting data out of a coroutine



Getting data out of a coroutine

```
Coroutine f1() {
  co_await TheAnswer{42};
}

TheAnswer::TheAnswer(int v)
:value_(v) {}
```

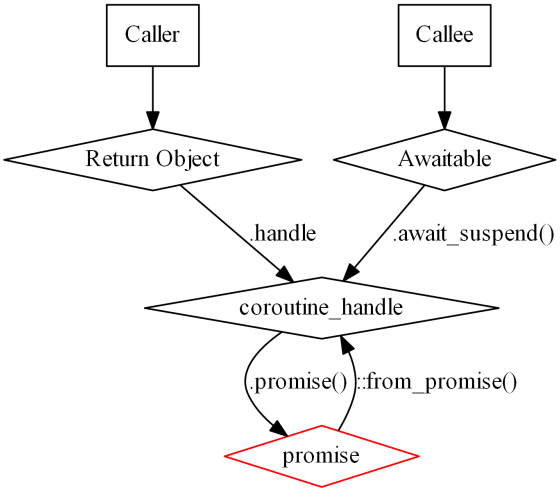
Getting data out of a coroutine



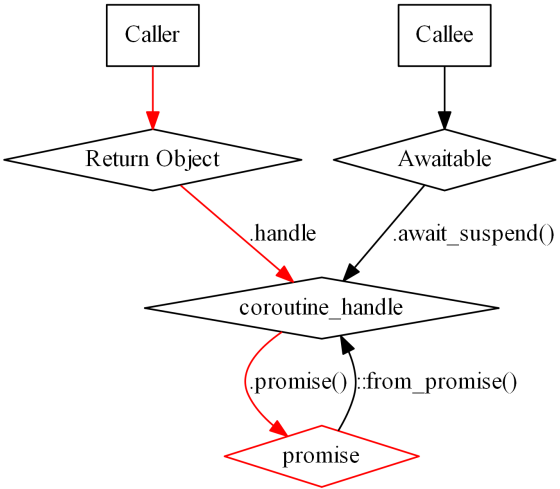
Getting data out of a coroutine

```
struct promise {
  // ...
  int value;
};

void TheAnswer::await_suspend(
       std::coroutine_handle<promise> h)
{
  h.to_promise().value = value_;
}
```

Getting data out of a coroutine



Getting data out of a coroutine

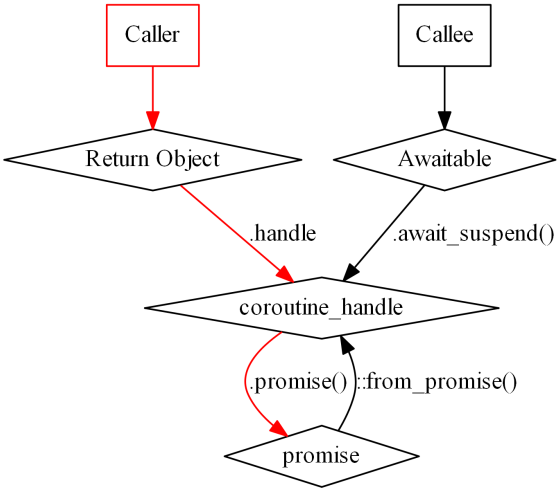


Getting data out of a coroutine

```
struct Coroutine {
  // ...
  std::coroutine_handle<promise> handle;
  int getAnswer() {
    return handle.promise().value;
  }
};
int main() {
  Coroutine c1 = f1();
  fmt::print("The answer is {}\n", c1.getAnswer());
}
```

Getting data into a coroutine

Getting data into a coroutine



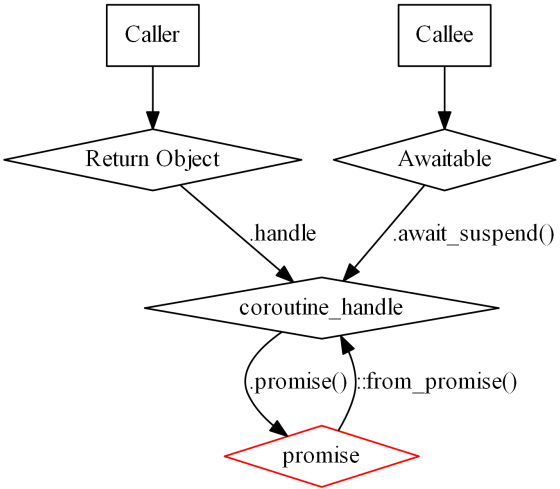
Getting data out of a coroutine

```
Coroutine f1() {
  int the_answer = co_await OutsideAnswer{};
}

int main() {
  Coroutine c1 = f1();
  c1.provide(42);
}
```

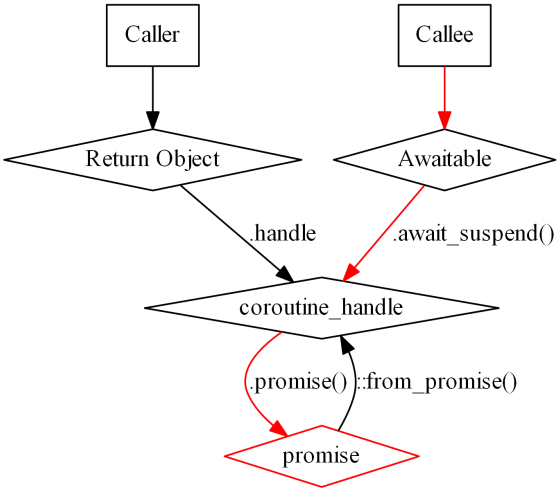
Getting data out of a coroutine

```
void Coroutine::provide(int the_answer) {
  handle.promise().value = the_answer;
  handle.resume();
}
```

Getting data into a coroutine



Getting data into a coroutine



Getting data out of a coroutine

```
struct OutsideAnswer {
  bool await_ready() { return false; }
  void await_suspend(std::coroutine_handle<promise> h) {
    handle = h;
  }
  int await_resume() {
    return handle.promise().value;
  }

  std::coroutine_handle<promise> handle;
};
```

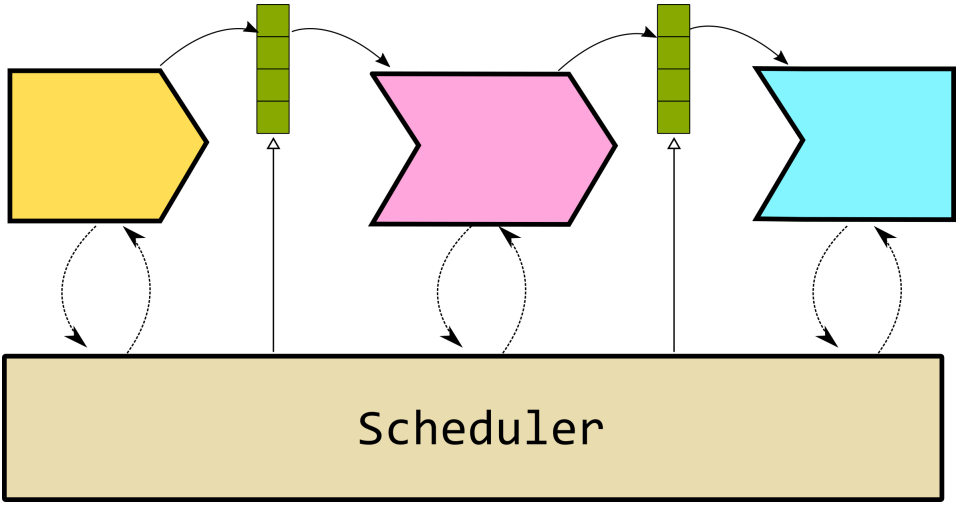
Pipelines with coroutines

```
while (true) {
  // ...
  AcquiredBuffer b_in =
    in_buffer.acquireFilledBuffer();

  // ...
  AcquiredBuffer b_out =
    out_buffer.acquireFreeBuffer();

  // ...
  step.process();
}
```

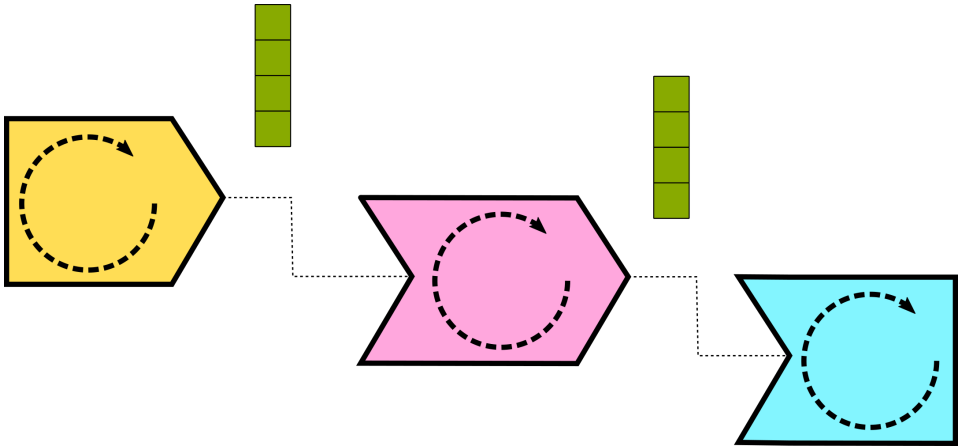
Pipelines with coroutines

```
while (true) {
  // ...
  AcquiredBuffer b_in =
    co_await AcquireFilledBuffer{ in_buffer };

  // ...
  AcquiredBuffer b_out =
    co_await AcquireFreeBuffer{ out_buffer };

  // ...
  step.process();
}
```

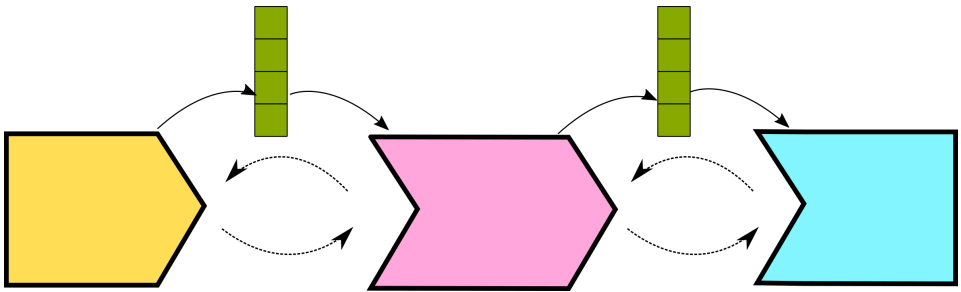
Pipelines with coroutines



Pipelines with coroutines



Pipelines with coroutines



Symmetric transfer

```
co_await Transfer{};

struct promise {
  // ...
  std::optional<std::coroutine_handle<promise>> other;
};

std::coroutine_handle<> Transfer::await_suspend(
              std::coroutine_handle<promise> me)
{
  return me.promise().other.value_or(me);
}
```

Pipelines with symmetric transfer

```
AcquiredBuffer b_out =
    co_await AcquireFreeBuffer{ out_buffer }

AcquireFreeBuffer::AcquireFreeBuffer(RingBuffer& b)
: buffer_(b)
{}

bool AcquireFreeBuffer::await_ready() {
  return buffer_.hasFreeBuffer();
}

std::coroutine_handle<>
  AcquireFreeBuffer::await_suspend(...) { ... }
```

Pipelines with symmetric transfer

Suspension behavior can be made configurable per step:

- Have big I/O buffers and asynchronous sources and sinks
- Non-I/O filters running on smaller buffers for better responsiveness
- Non-I/O always runs to completion and only suspends when exhausting its input or otput buffers
- $\Rightarrow$ I/O latency is being hidden

Final result

```
step(FileSource{ from_filename }, buffer_fin } )
| step(FilterDeflate{}, buffer_deflate )
| step(FileSink{ to_filename });
```

Allows independent adjustment of

- Processing step implementation
- Scheduling logic
- Buffer sizes

Wrapping up...

- Data processing is difficult, but can be made very easy when a framework takes care of the hard stuff
- Coroutines provide powerful tools for writing such frameworks
- Manipulating control flow of tasks through coroutines is extremely powerful
- However, coroutines are a complex tool with many sharp edges. Development experience in the C++ community is limited
- Try them out, get your brain mangled today!

References

- Coroutines on cppreference
- Asymmetric Transfer - Lewis Baker's blog on coroutines
- The Old New Thing - Raymond Chen's blog [1] [2]
- Eric Niebler: Introducing the Ranges TS (code::dive 2017)
- Coroutines TS (N4760)

Thanks for your attention.

ComicSansMS / @DerGhulbus