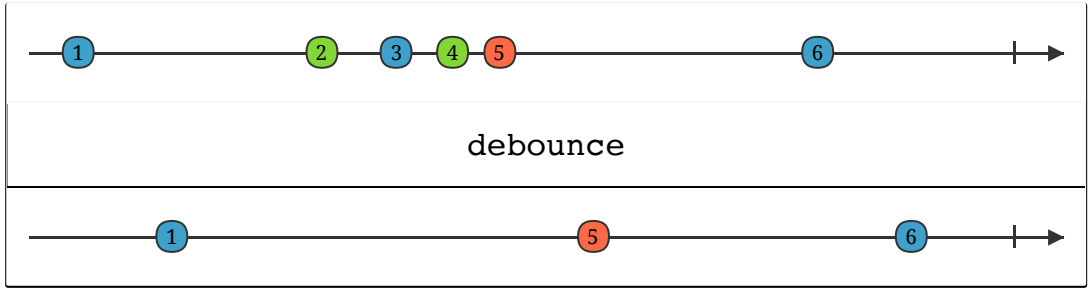
Adding Async Algorithms to std

algorithms for values distributed in time



me

I love code

me

I work at microsoft

what is the goal here?

- additive to existing (stl, rangev3, parrallel_stl)
- scan dense material
- derive requirements from selected algorithms

What are examples of values distributed in time?

ints distributed in time

code (rxcppv3)

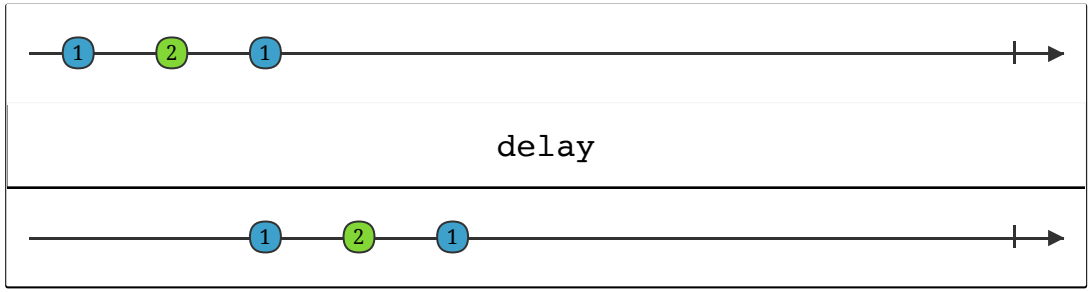
```
auto threeeven = copy_if(even) |
  take(3) |
  delay(makeStrand, 1s);

intervals(makeStrand, steady_clock::now(), 1s) |
  threeeven |
  as_interface<long>() |
  finally([](){cout << "caller stopped" << endl;}) |
  printto(cout) |
  start<destruction>(subscription{}, destruction{});
```

output (emscripten)

```
caller stopped
0 - 0.6s - destructed
```

delay

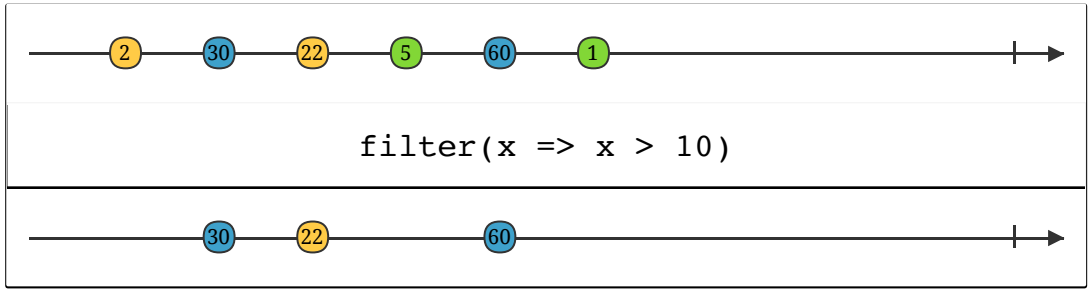


ux events distributed in time

code (coroutine algorithms)

```
std::future<void> AddVisuals(CoreWindow window, VisualCollection visuals)
{
  for co_await (auto move : window.co_PointerPressed() |
    transform([](auto press) -> float2 {return press.args.CurrentPoint().Position(); }) |
    filter([](auto point) {return !VisualAndOffsetFromPoint(visuals, point).first; }))
  {
    AddVisual(visuals, point);
  }
}
```

copy_if

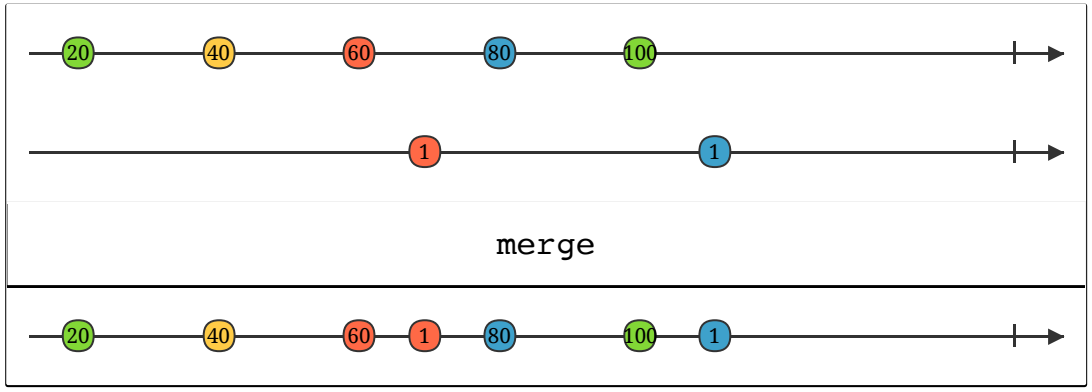


ux events distributed in time

code (coroutine algorithms)

```
std::future<void> MoveVisuals(CoreWindow window, VisualCollection visuals) {
  for co_await (auto move : window.co_PointerPressed() |
    transform([](auto press) -> float2 {return press.args.CurrentPoint().Position(); }) |
    transform([=](auto point) {return VisualAndOffsetFromPoint(visuals, point); }) |
    filter([](auto selected) {return !!selected.first; }) |
    transform([=](auto selected) {
      MoveToTop(visuals, selected.first);
      return window.co_PointerMoved() |
        transform([](auto move) -> float2 {return move.args.CurrentPoint().Position(); }) |
        transform([=](auto point) {
          auto to = float2{ point.x + selected.second.x, point.y + selected.second.y };
          return std::make_pair(selected.first, to);
        }) |
        take_until(window.co_PointerReleased());
    }) |
    merge()) {
    move.first.Offset({ move.second.x, move.second.y, 0.0f });
  }
}
```

merge



ux events distributed in time

code (rxcpp)

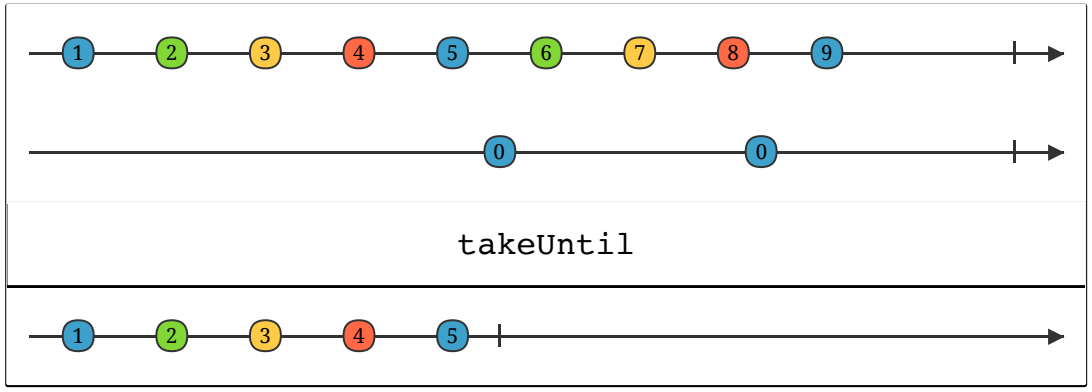
```
auto down$ = mousedown$("#window");
auto up$ = mouseup$("#window");
auto move$ = mousemove$("#window");

down$ |
  flat_map([=](MouseEvent){
    return move$ |
      take_until(up$) |
      map([](MouseEvent){return 1;}) |
      start_with(0) |
      sum();
  }) |
  map(
    [](int c){
      return to_string(c) + " moves while mouse down";
    }) |
  subscribe(println(cout));
```

output (emscripten)

```
2 moves while mouse down
```

take_until



packets of bytes distributed in time

code (rxcpp)

```
asyncReadBytes() |
  tap(printVectorOfBytes) |
  concat_map(vectorOfStringsFromVectorOfBytes) |
  group_by(groupFromString) |
  flat_map(appendGroupStrings) |
  subscribe(println(cout));
```

output (emscripten)

http requests distributed in time

code (rxcpp)

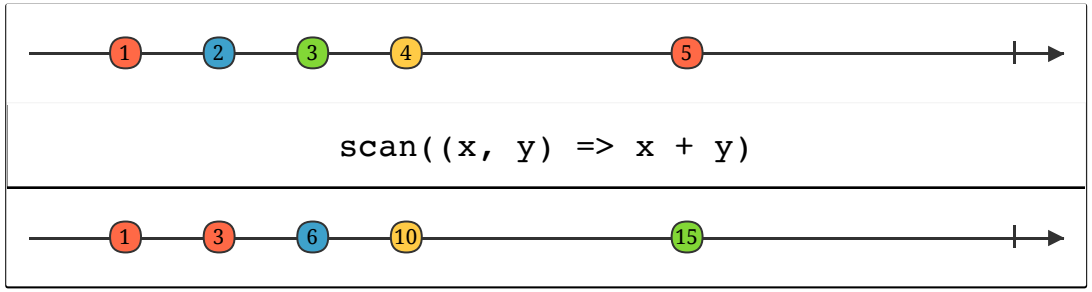
```
struct data { int size; string firstLine;};
struct model { map<string, data> store; };

httpGet("https://aka.ms/rxcppreadme") |
  flat_map([](response_t r) {
    return r.progress() |
      combine_latest(
        [=](progress_t p, vector<uint8_t> d){
          return make_tuple(r.url(), p, d);
        },
        r.load()) |
      scan(
        model{},
        updateModelFromTuple);
  }) |
  subscribe(println(cout));
```

output (emscripten)

```
README.md, 0
README.md, 0
README.md, 8105
README.md, 8105
The Reactive Extensions for Native
(__RxCpp__) is a library for
composing asynch
```

scan



why alogrithms?

- documented
- stable
- optimized
- descriptive

what do algorithms operate on?

What ways can a sequence be delivered?

What ways can a sequence be delivered?

in space

- vector of mouse positions
- generator of mouse positions

```
using mouseMoves = vector<tuple<int,int>>;
```

0, 0	100, 100	200, 200	300, 300	400, 400

```
auto mouseMoves(int start, int end)
    -> std::generator<tuple<int, int>> {
    for(;start != end; ++start){
        auto position = start * 100;
        co_yield make_tuple(position, position);
    }
}
```

What ways can a sequence be delivered?

in time

- mouse move events
- network packets

```
auto window::mouseMoves()
    -> co_generator<tuple<int, int>> {
    for co_await(auto event : events()) {
        if (event.id == MOUSEMOVE) {
            co_yield mousePositionFrom(event);
        }
    }
}

auto socket::bytes()
    -> co_generator<vector<byte>> {
    vector<byte> out;
    while (out = co_await read(. . .)) {
        co_yield out;
    }
}
```

ReactiveExtensions

- algorithms for values distributed in time
- implementations for many languages

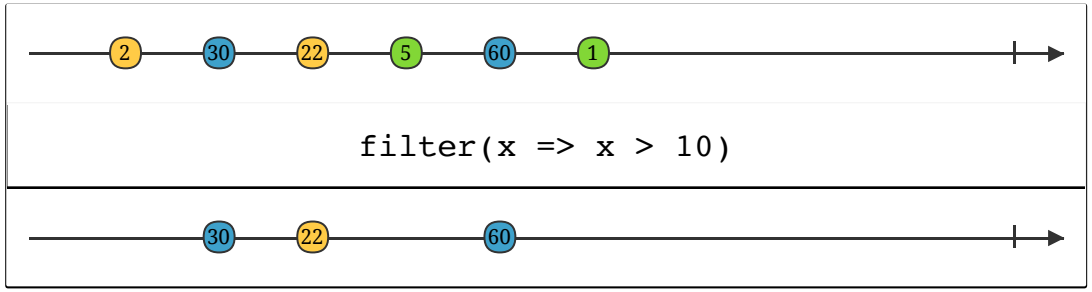
what algorithms are supported in rxcpp?

rxcpp uses cpp11 in vs2013, vs2015, clang and gcc

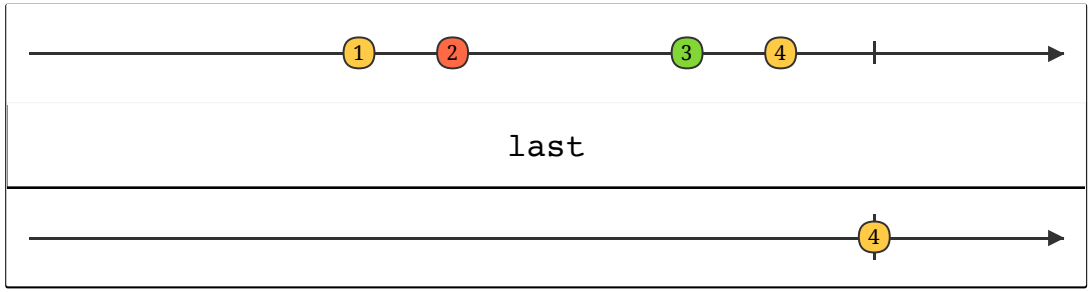
- Combining
  - amb, buffer, combine_latest, concat, concat_map, flat_map, group_by, merge , switch_if_empty, switch_on_next, window, window_toggle, with_latest_from, zip
- Transforming
  - delay, map, pairwise, on_error_resume_next, reduce, scan
- Filtering
  - default_if_empty, distinct, distinct_until_changed, element_at, ignore_elements, take, take_last, take_until, skip, skip_last, skip_until, sample, debounce, filter
- Others
  - all, contains, exists, observe_on, publish, repeat, replay, retry, sequence_equal, subscribe_on, tap, time_interval, timeout, timestamp

what algorithms will help define the concepts?

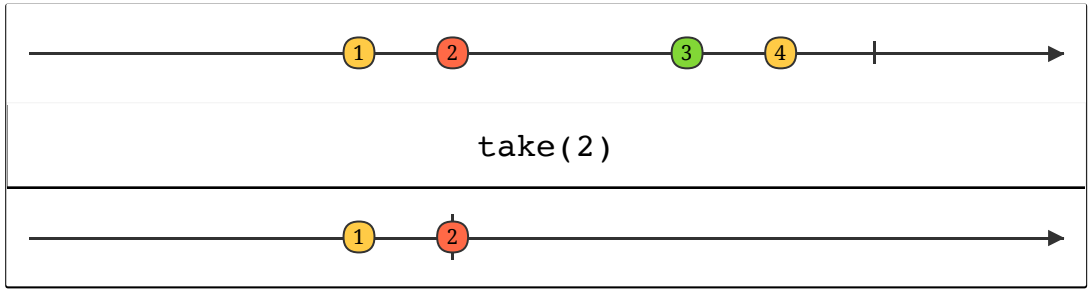
copy_if



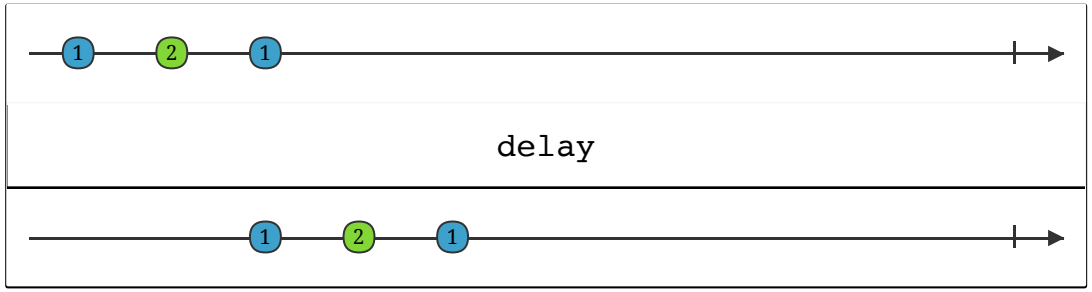
last_or_default



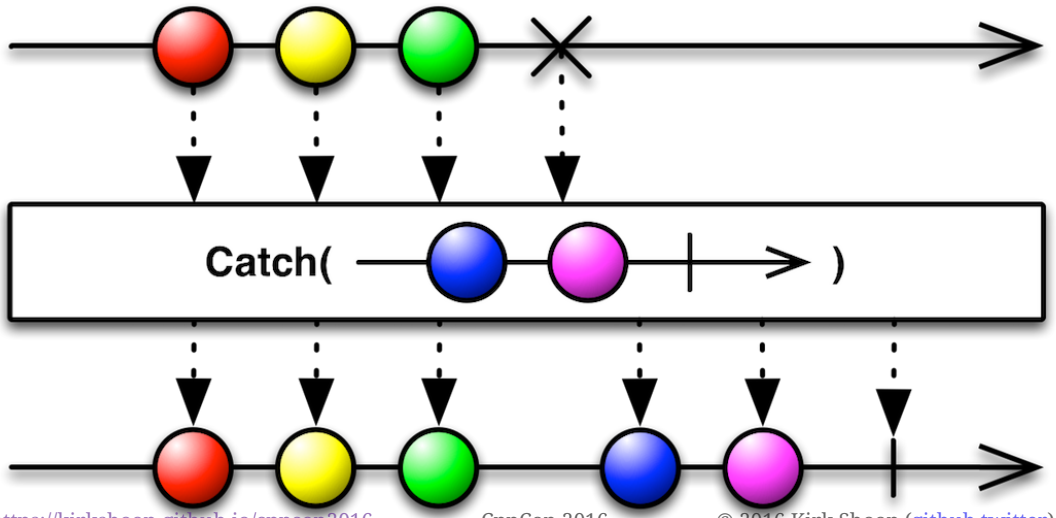
take



delay



resume_error



what features are required to build the desired algorithms?

rxcppv3 is a proof-of-concept in cpp14 that will be used to explore the requirements

what are the minimum features?

sequence concepts

```
struct observable {
    void bind(observer);
};

struct observer {
    template<class T>
    void next(T);
};

struct lifter {
    observer lift(observer);
};
```

sequence implementations

```
const auto ints = [](auto first, auto last){
  return make_observable([=](auto r){
    for(auto i = first;; ++i){
      r.next(i);
      if (i == last) break;
    }
  });
};

const auto copy_if = [](auto pred){
  return make_lifter([=](auto r){
    return make_observer(r, [=](auto& r, auto v){
        if (pred(v)) r.next(v);
    });
  });
};
```

push sequence

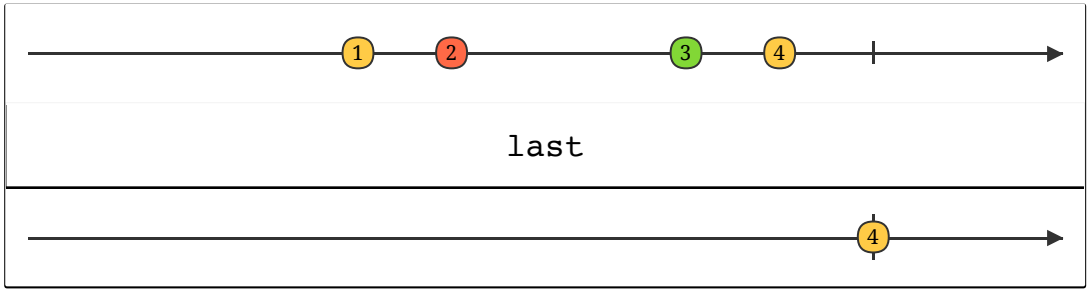
code

```
ints(0, 9) |
  copy_if(even) |
  printto(cout) |
  start<destruction>(subscription{}, destruction{});
```

output (emscripten)

```
0 - 0.0s - 0
0 - 0.0s - 2
0 - 0.0s - 4
0 - 0.0s - 6
0 - 0.0s - 8
0 - 0.0s - 5 values received - done!
0 - 0.0s - destructed
```

what needs to change to support last_or_default?



sequence concepts

```
struct observer {
    template<class T>
    void next(T);
    void complete();
};
```

sequence implementations

```
const auto last_or_default = [](auto def){
  return make_lifter([=](auto scbr){
    return make_subscriber([=](auto ctx){
      auto r = scbr.create(ctx);
      using last_t = std::decay_t<decltype(def)>;
      auto last = make_state<last_t>(ctx.lifetime, def);
      return make_observer(r, r.lifetime,
        [last](auto& , auto v){
          last.get() = v;
        },
        [last](auto& r){
          r.next(last.get());
          r.complete();
        });
    });
  });
};
```

what needs to change to support last_or_default?

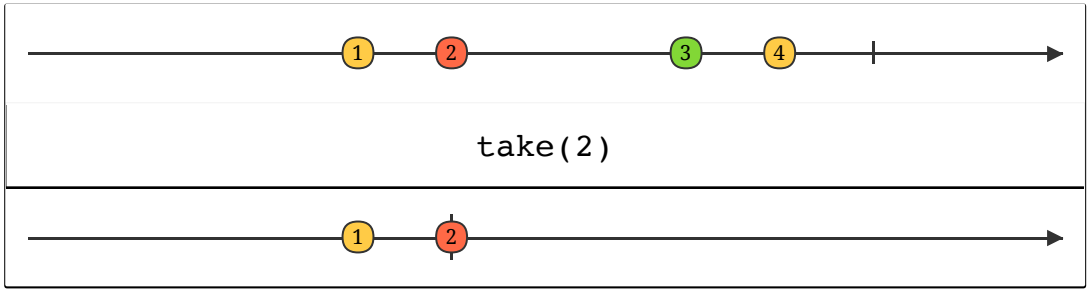
code

```
ints(0, 100000) |
  copy_if(even) |
  last_or_default(42) |
  printto(cout) |
  start<destruction>(subscription{}, destruction{});
```

output (emscripten)

```
0 - 0.0s - destructed
```

what needs to change to support take?



what features are needed to support asynchronous lifetime and cancellation?

- signal for explicit cancellation of lifetime graph
- registration for cancellation signal
- nested lifetime graph
- make_shared equivalent for asynchronous lifetime

sequence concepts

```
struct subscription
{
  bool is_stopped();
  void stop(); // signal

  // registration
  void insert(function<void()> stopper);

  // nested
  void insert(const subscription& s);
  void erase(const subscription& s);

  // make_shared<Payload>
  template<class Payload, class... ArgN>
  state<Payload> make_state(
    ArgN... argn);
};

template<class Payload>
struct context {
    subscription lifetime;
    Payload& get();
};
```

sequence concepts

```
struct starter {
    template<class Payload>
    subscription start(context<Payload>);
};

struct subscriber {
    template<class Payload>
    observer create(context<Payload>);
};

struct observable {
    starter bind(subscriber);
};

struct lifter {
    subscriber lift(subscriber);
};

struct adaptor {
    observable adapt(observable);
};
```

sequence implementations

```
const auto take = [](int n){
  return make_adaptor([=](auto source){
    return make_observable([=](auto scrb){
        return source.bind(
          make_subscriber([=](auto ctx){
            auto r = scrb.create(ctx);
            auto remaining = make_state<int>(r.lifetime, n);
            auto lifted = make_observer(r, r.lifetime,
              [remaining](auto& r, auto v){
                r.next(v);
                if (--remaining.get() == 0) {
                    r.complete();
                }
              });
            if (n == 0) {
              lifted.complete();
            }
            return lifted;
          }));
    });
  });
};
```

what needs to change to support take?

code

```
ints(0, 9) |
  copy_if(even) |
  take(3) |
  printto(cout) |
  start<destruction>(subscription{}, destruction{});
```

output (emscripten)

```
0 - 0.0s - destructed
0 - 0.0s - 0
0 - 0.0s - 2
0 - 0.0s - 4
0 - 0.0s - 3 values received - done!
0 - 0.0s - destructed
```

what needs to change to support failure?

sequence concepts

```
struct observer {
    template<class T>
    void next(T);

    template<class E>
    void error(E);

    void complete();
};
```

what needs to change to support failure?

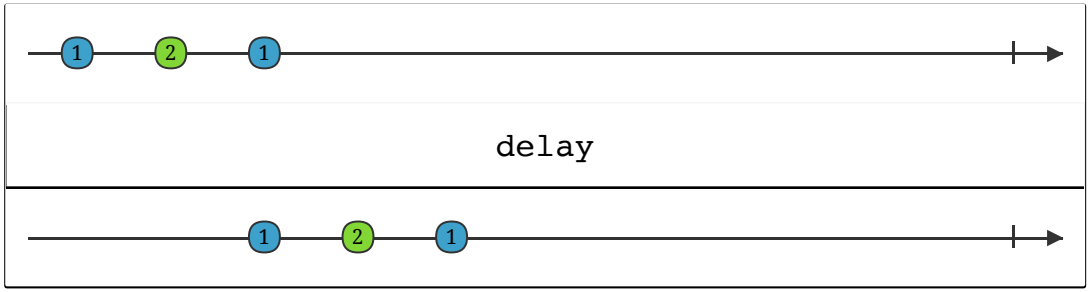
code

```
ints(0, 9) |
  copy_if(always_throw) |
  take(count) |
  printto(cout) |
  start<destruction>(subscription{}, destruction{});
```

output (emscripten)

```
0 - 0.0s - always throw!
0 - 0.0s - destructed
```

what needs to change to support delay?



what needs to change to support delay?

defer concepts

```
template<class Clock>
struct strand {
    subscription lifetime;

    Clock::time_point now();
    void defer_at(Clock::time_point, observer);
};

template<class Payload, class Clock>
struct context {
    subscription lifetime;

    Clock::time_point now();
    void defer_at(Clock::time_point, observer);

    Payload& get();
};
```

sequence concepts

```
struct starter {
    template<class Payload, class Clock>
    subscription start(context<Payload, Clock>);
};

struct subscriber {
    template<class Payload, class Clock>
    observer create(context<Payload, Clock>);
};
```

what needs to change to support delay?

code

```
intervals(makeStrand, steady_clock::now(), 1s) |
  printproduced(cout) |
  delay(makeStrand, 1500ms) |
  take(3) |
  printto(cout) |
  start<destruction>(subscription{}, destruction{});
```

output (emscripten)

```
0 - 0.0s - destructed
```

what needs to change to support testing?

what needs to change to support testing?

```
struct virtual_clock {
    static bool is_steady() const;
    time_point now() const;
    void now(time_point at);
};
```

```
struct test_loop {
  void call(item_type& next) const;
  void step(typename clock_type::duration d) con
  void run() const;
  makeStrand make() const;
};
```

what needs to change to support testing?

```
struct recorded {
  lifetime_record lifespan(
    time_point start = duration{200},
    time_point stop = duration{1000});

  marble_record next(time_point at, T v);
  marble_record error(time_point at, error_t e);
  marble_record complete(time_point at);

  observable hot(std::vector<marble_record> m);
  observable cold(std::vector<marble_record> m);

  //...
};
```

```
  //...

  struct test_result {
      time_point origin;
      lifetime_record lifespan;
      map<string, vector<marble_record>> marbles
  };
  lifter record(string key) const;
  template<class... TN>
  terminator test(test_loop<TN...>& tl,
    lifetime_record l = lifetime_record{});
};
```

what needs to change to support testing?

code

```
rx::test_loop<> loop;
rx::recorded<int> on;
auto tr = on.hot({
        on.next(on.origin() + 1s * 1, 0),
        on.next(on.origin() + 1s * 2, 1),
        on.next(on.origin() + 1s * 3, 2),
        on.next(on.origin() + 1s * 4, 3),
        on.next(on.origin() + 1s * 5, 4)
    }) |
    printout(cout, "produced") |
    on.record("produced") |
    delay(loop.make(), 1500ms) |
    take(3) |
    printout(cout, "") |
    on.record("emitted") |
    on.test(loop);
loop.run();
```

output (emscripten)

```
0 - 1.0s - 0 produced
0 - 2.0s - 1 produced
0 - 2.5s - 0
0 - 3.0s - 2 produced
0 - 3.5s - 1
0 - 4.0s - 3 produced
0 - 4.5s - 2
Elapsed: 0.0
SUCCEEDED
SUCCEEDED
```

what needs to change to support testing?

output (emscripten)

output (emscripten)

```
0 - 1.0s - 0 produced
0 - 2.0s - 1 produced
0 - 2.5s - 0
0 - 3.0s - 2 produced
0 - 3.5s - 1
0 - 4.0s - 3 produced
0 - 4.5s - 2
Elapsed: 0.0
SUCCEEDED
SUCCEEDED
```

what needs to change to support testing?

record test result

```
rx::test_loop<> loop;
rx::recorded<int> on;
auto tr = on.hot({
        on.next(on.origin() + 1s * 1, 0),
        on.next(on.origin() + 1s * 2, 1),
        on.next(on.origin() + 1s * 3, 2),
        on.next(on.origin() + 1s * 4, 3),
        on.next(on.origin() + 1s * 5, 4)
    }) |
    printout(cout, "produced") |
    on.record("produced") |
    delay(loop.make(), 1500ms) |
    take(3) |
    printout(cout, "") |
    on.record("emitted") |
    on.test(loop);
loop.run();
```

assert test succeeded

```
auto expected = on.expected({
    on.next(origin() + 1s * 1 + 1500ms, 0),
    on.next(origin() + 1s * 2 + 1500ms, 1),
    on.next(origin() + 1s * 3 + 1500ms, 2),
    on.complete(origin() + 1s * 3 + 1500ms)
});
if (tr.get().marbles["emitted"] == expected) {
    cout << "SUCCEEDED" << endl;
} else {
    cout << "FAILED" << endl;
    cout << "Actual:" << endl;
    cout << tr.get().marbles["emitted"] << endl;
    cout << "Expected:" << endl;
    cout << expected << endl;
}
```

what needs to change to support testing?

failing code

```
rx::test_loop<> loop;
rx::recorded<int> on;
auto tr = on.hot({
        on.next(on.origin() + 1s * 1, 0),
        on.next(on.origin() + 1s * 2, 1),
        on.next(on.origin() + 1s * 3, 2),
        on.next(on.origin() + 1s * 4, 3),
        on.next(on.origin() + 1s * 5, 4)
    }) |
    printout(cout, "produced") |
    on.record("produced") |
    delay(loop.make(), 1500ms) |
    take(1) |
    printout(cout, "") |
    on.record("emitted") |
    on.test(loop);
loop.run();
```

output (emscripten)

```
0 - 1.0s - 0 produced
0 - 2.0s - 1 produced
0 - 2.5s - 0
Elapsed: 0.0
FAILED
Actual:
[next@2500{0}, complete@2500{}, ]
Expected:
[next@2500{0}, next@3500{1},
next@4500{2}, complete@4500{}, ]
FAILED
Actual:
{200, 2500}
Expected:
{200, 4500}
```

Requirements

- next, complete, error.
- lifetime, allocation.
- cancellation, scheduling.
- virtual-time, testing

What if these algorithms are implemented using the coroutine proposal?

```
co_value_generator<SelectValue> transform(Source source, Selector select) {
  for co_await (auto&& v : source) {
    co_yield select(v);
  }
}

co_value_generator<SourceValue> concat(Source source) {
  for co_await (auto&& s : source) {
    for co_await (auto&& v : s) {
      co_yield v;
    }
  }
}
```

What if these algorithms are implemented using the coroutine proposal?

```
struct merge_value_promise : co_generator_promise<T>
{
  // >200 lines of code

  merge_source_awaiter push(Source s) const {
    auto& p = co_await merge_source_awaiter::get();
    p.bind(this, canceled);
    for co_await (auto& v : s) {
      co_yield v;
    }
  }
};

co_generator<merge_value_promise<SourceValue>> merge(Source source) {
  auto& p = co_await merge_value_promise<SourceValue>::get();
  co_await p.caller_awaiter(nullptr);
  for co_await (auto&& s : source) {
    p.push(std::move(s));
  }
}
```

What if these algorithms are implemented using the coroutine proposal?

transform usage

```
future<void> print0to9doubled() {
  for co_await(auto v : ints(0, 9) | transform([](int v){return v * 2;})) {
    cout << v << endl;
  }
}
```

What if these algorithms are implemented using the coroutine proposal?

sequence concepts

```
template <typename P>
struct co_generator
{
  using iterator = co_iterator<value_type>;

  co_iterator_awaiter<value_type> begin() const;

  iterator end() const;

  co_generator_promise<T> const * p;
};
```

sequence concepts

```
template <typename T>
struct co_iterator
  : std::iterator<std::input_iterator_tag, T>
{
  // end iterator
  co_iterator(nullptr_t) : m_p(nullptr);
  // iterator
  co_iterator(co_generator_promise<T> const & p)

  co_inc_awaiter<T> operator++();

  bool operator==(co_iterator const &rhs) const;

  T &operator*();
  T *operator->();

  co_generator_promise<T> const * m_p;
};
```

What if these algorithms are implemented using the coroutine proposal?

sequence concepts

```
template <typename T>
struct co_iterator_awaiter
{
  bool await_ready();

  void await_suspend(
    const coroutine_handle<>& handle);

  co_iterator<T> await_resume();

  co_generator_promise<T> const * m_p;
};
```

sequence concepts

```
template <typename T>
struct co_inc_awaiter
{
  bool await_ready();

  void await_suspend(
    const coroutine_handle<>& handle);

  co_iterator<T>& await_resume();

  co_iterator<T>* m_it;
};
```

complete.

questions?