At the beginning of time…

WORKING WITH COLLECTIONS IN C++

WORKING WITH COLLECTIONS IN C++

-∞ → Now


```
for (
```

STL algorithms

ranges

Before we realised STL algorithms were a thing

When we realised they were important after all

Modern C++

STL ALGORITHMS

Applying a function

```
auto const inputs = std::vector<int>{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
auto results = std::vector<int>{};

std::transform(begin(inputs), end(inputs),
               back_inserter(results),
               [](int i){ return i * 2; });
```

Filtering

```
auto const inputs = std::vector<int>{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
auto results = std::vector<int>{};

std::copy_if(begin(inputs), end(inputs),
             back_inserter(results),
             [](int i){ return i % 3 == 0; });
```

transform_if

?

RANGES

transform_if

```
auto const inputs = std::vector<int>{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::vector<int> const results = inputs
    | ranges::view::filter([](int i){ return i % 3 == 0; })
    | ranges::view::transform([](int i){ return i * 2; });
```

transform-filter

```
auto const inputs = std::vector<int>{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::vector<int> const results = inputs
    | ranges::view::transform([](int i){ return i * 2; })
    | ranges::view::filter([](int i){ return i % 3 == 0; });
```

RANGES

transform-filter

```
auto const inputs = std::vector<int>{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::vector<int> const results = inputs
    | ranges::view::transform([](int i){ return i * 2; })
    | ranges::view::filter([](int i){ return i % 3 == 0; });
```

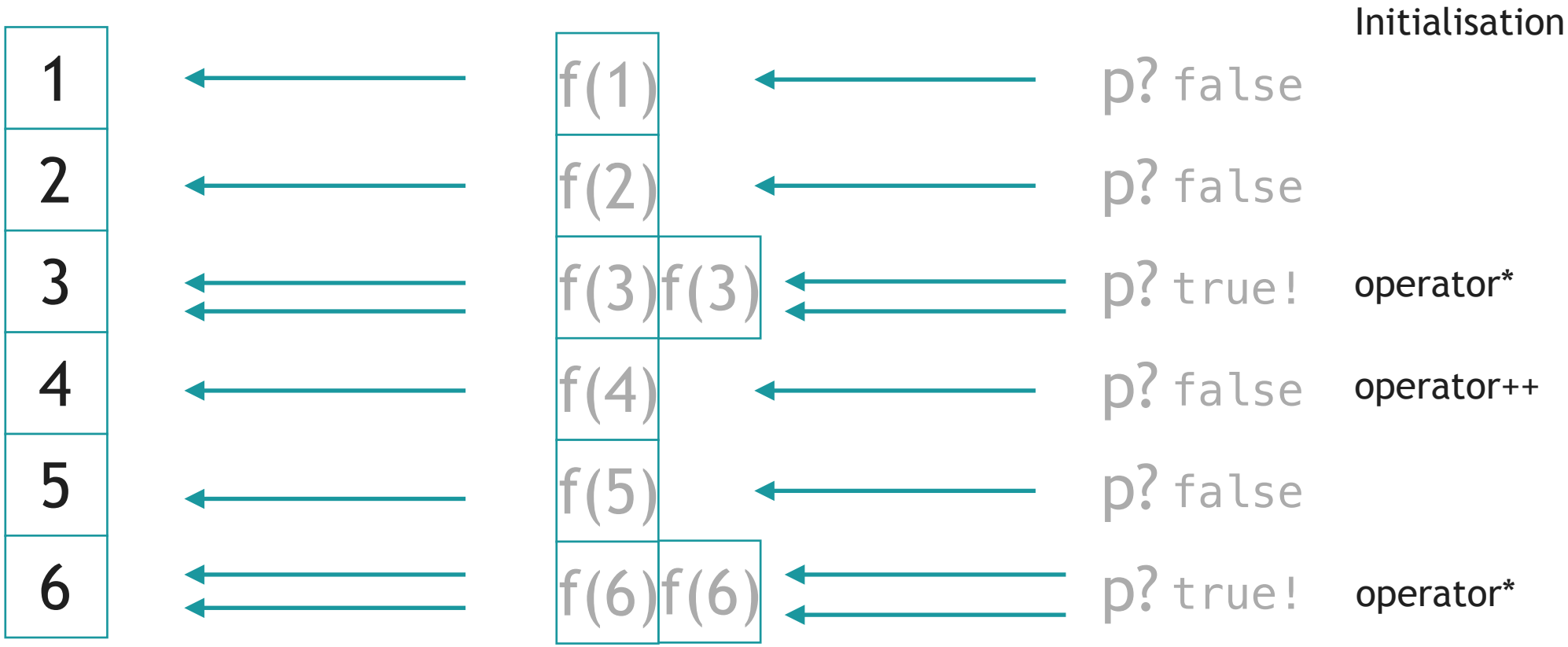
RANGES

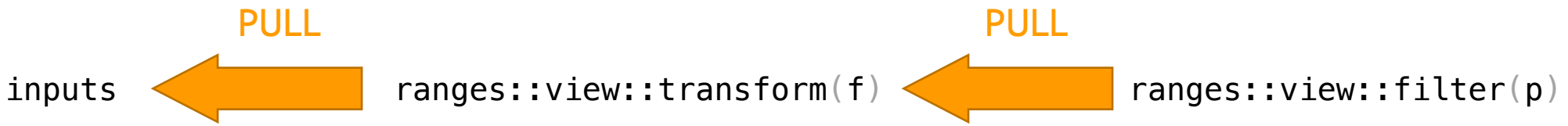
transform-filter

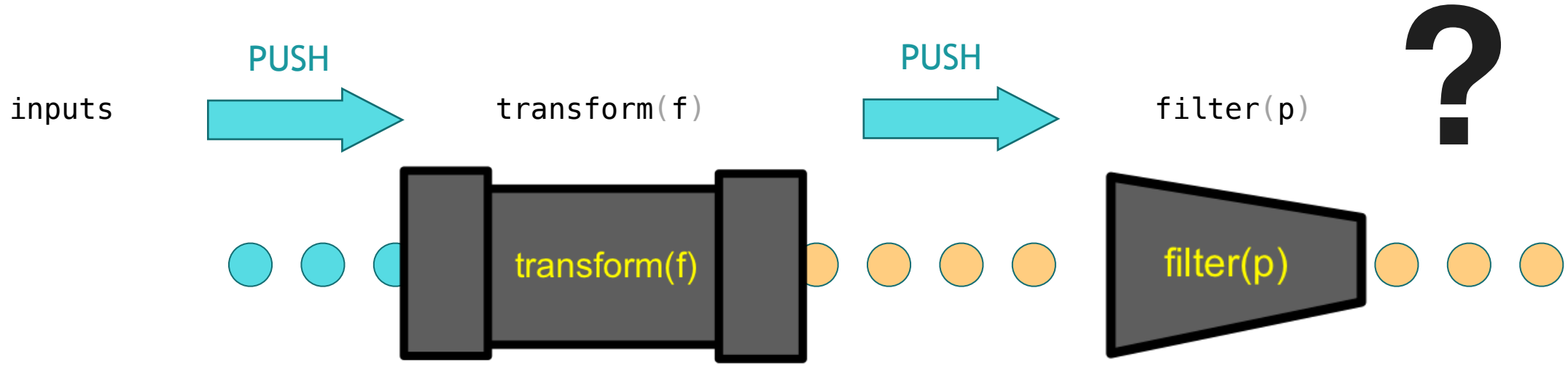
```
auto const inputs = std::vector<int>{1, 2, 3, 4, 5, 6};

std::vector<int> const results = inputs
    | ranges::view::transform([](int i){ cout << i << '\n'; return i * 2; })
    | ranges::view::filter([](int i){ return i % 3 == 0; });
```

```
1
2
3
3
4
5
6
6
```

```
inputs  |  ranges::view::transform(f)  |  ranges::view::filter(p)
```

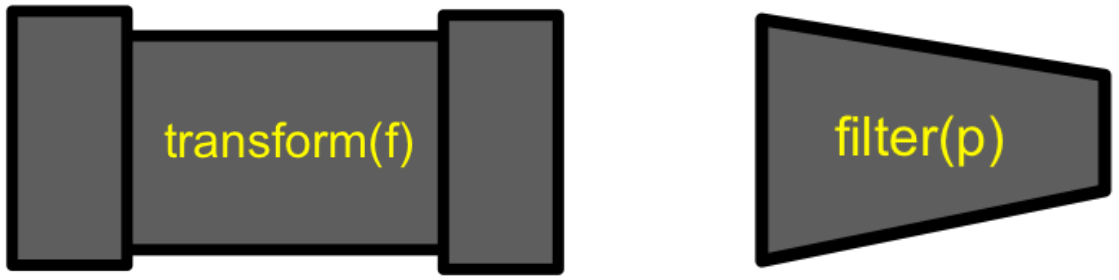



PULL
PULL
inputs
ranges::view::transform(f)
ranges::view::filter(p)

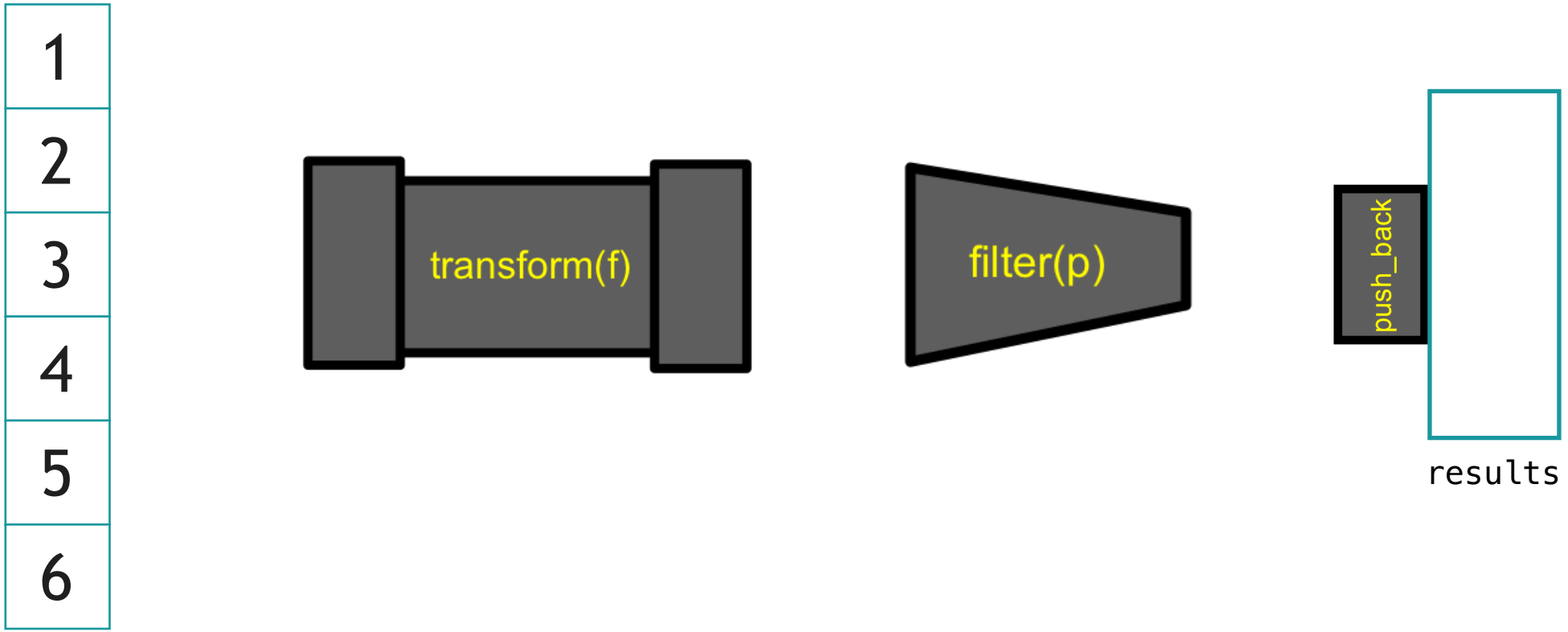

PUSH
PUSH
inputs
transform(f)
filter(p)
?
transform(f)
filter(p)

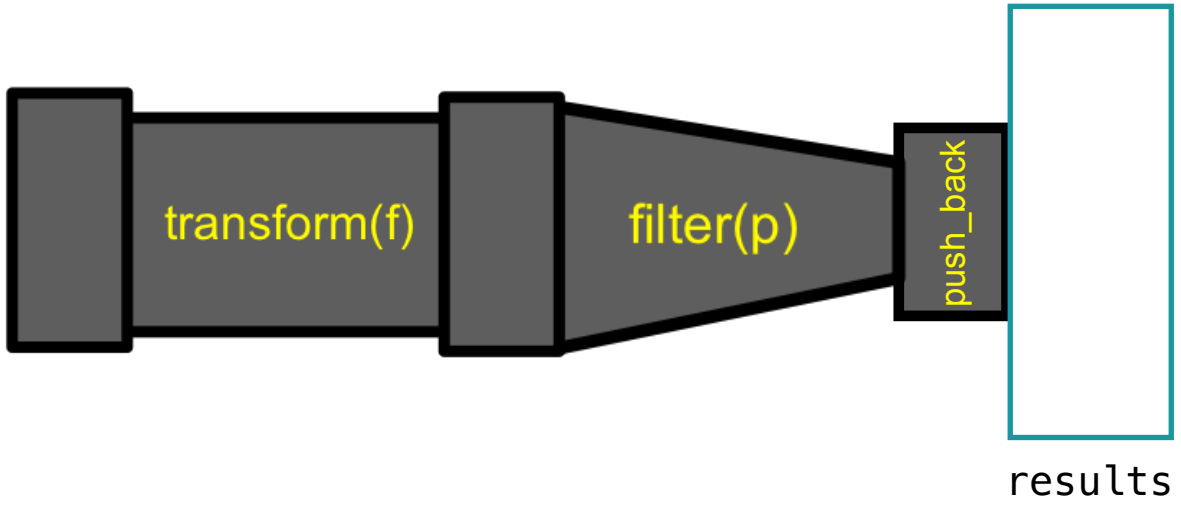
```
inputs  >>=  pipes::transform(f)  >>=  pipes::filter(p)  >>=  pipes::push_back(results)
```

1
2
3
4
5
6



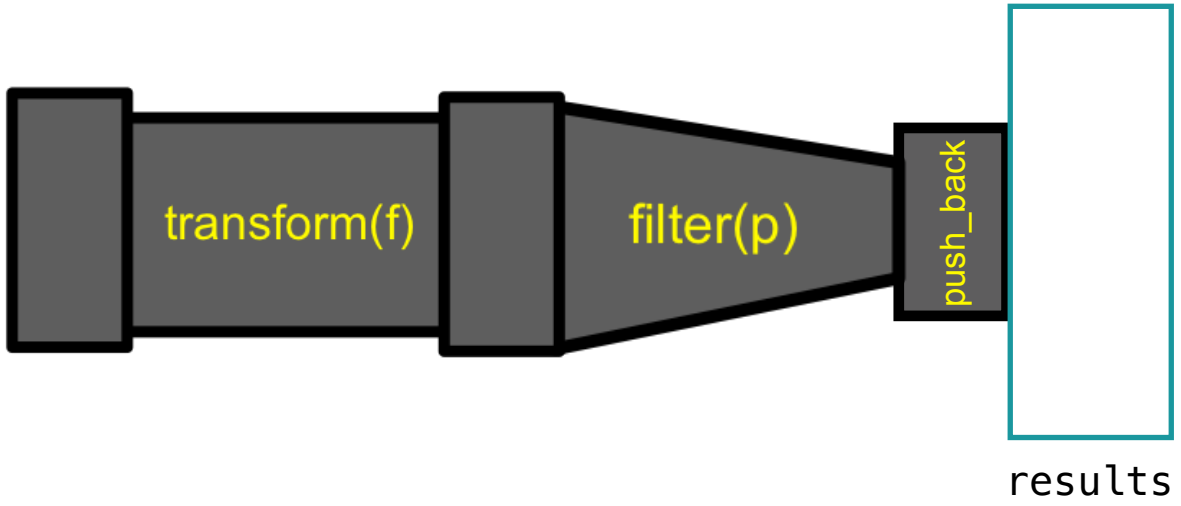
```
inputs    >>=    pipes::transform(f)    >>=    pipes::filter(p)    >>=    pipes::push_back(results)
```






```
auto const inputs = std::vector<int>{1, 2, 3, 4, 5, 6};
auto results = std::vector<int>{};

inputs >>= pipes::transform([](int i){ return i * 2; })
       >>= pipes::filter([](int i){ return i % 3 == 0; })
       >>= pipes::push_back(results);
```

github.com/JoBoccara/pipes

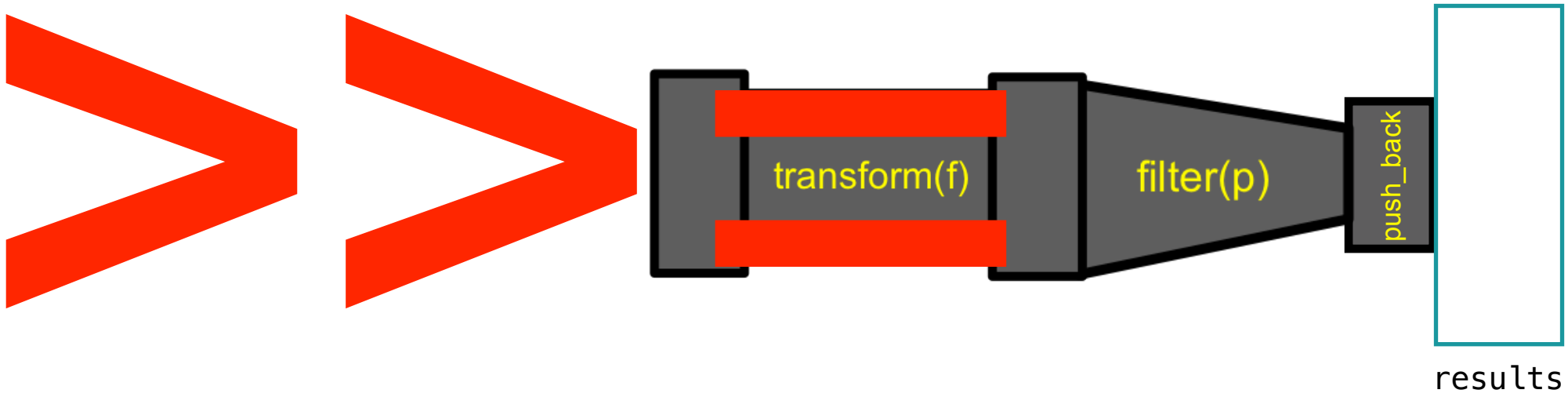


Why operator>>= ?

```
auto const inputs = std::vector<int>{1, 2, 3, 4, 5, 6};
auto results = std::vector<int>{};

inputs >>= pipes::transform([](int i){ cout << i << '\n'; return i * 2; })
       >>= pipes::filter([](int i){ return i % 3 == 0; })
       >>= pipes::push_back(results);
```

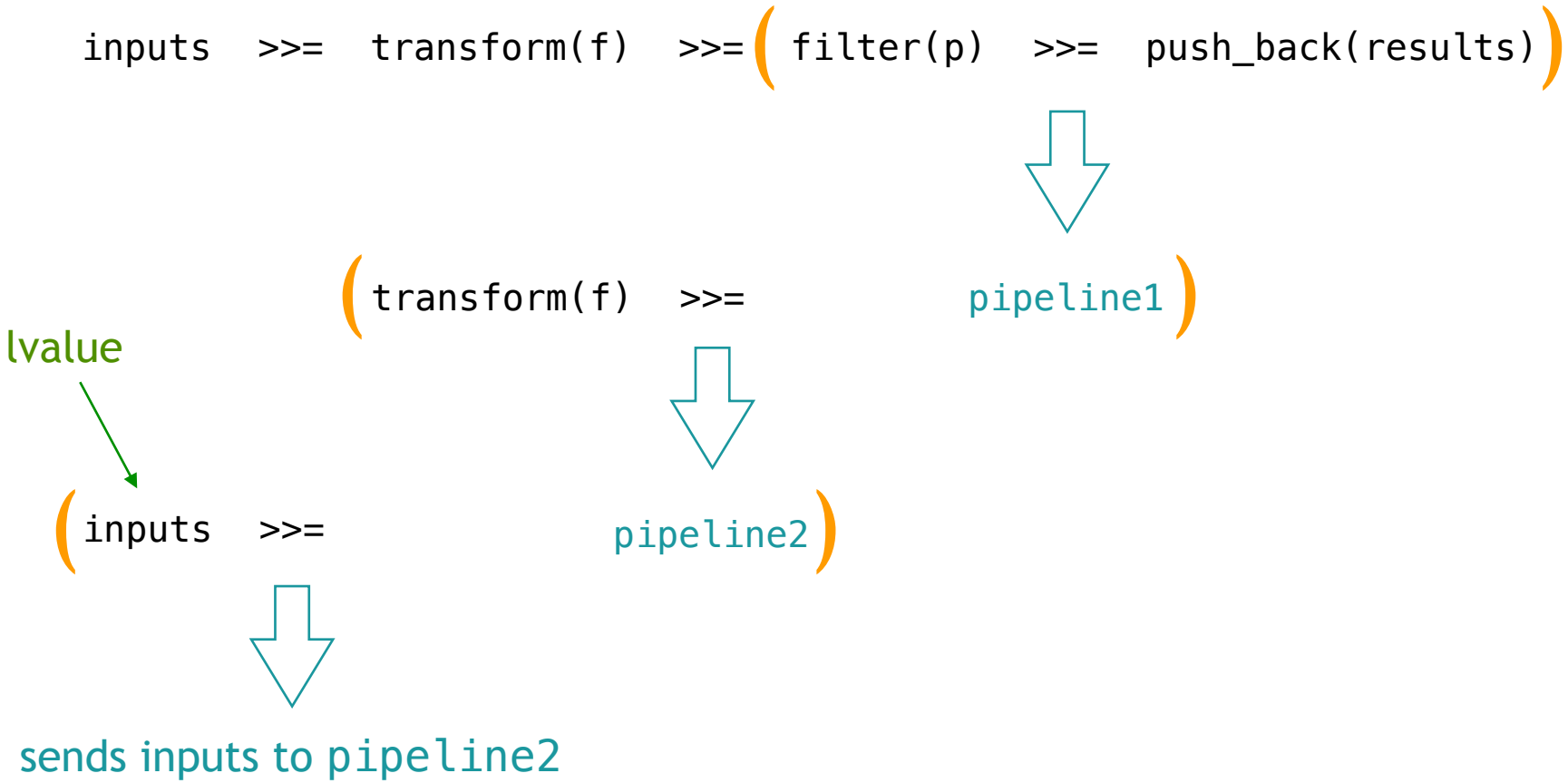
```
1
2
3
4
5
6
```

Why operator>>= ?

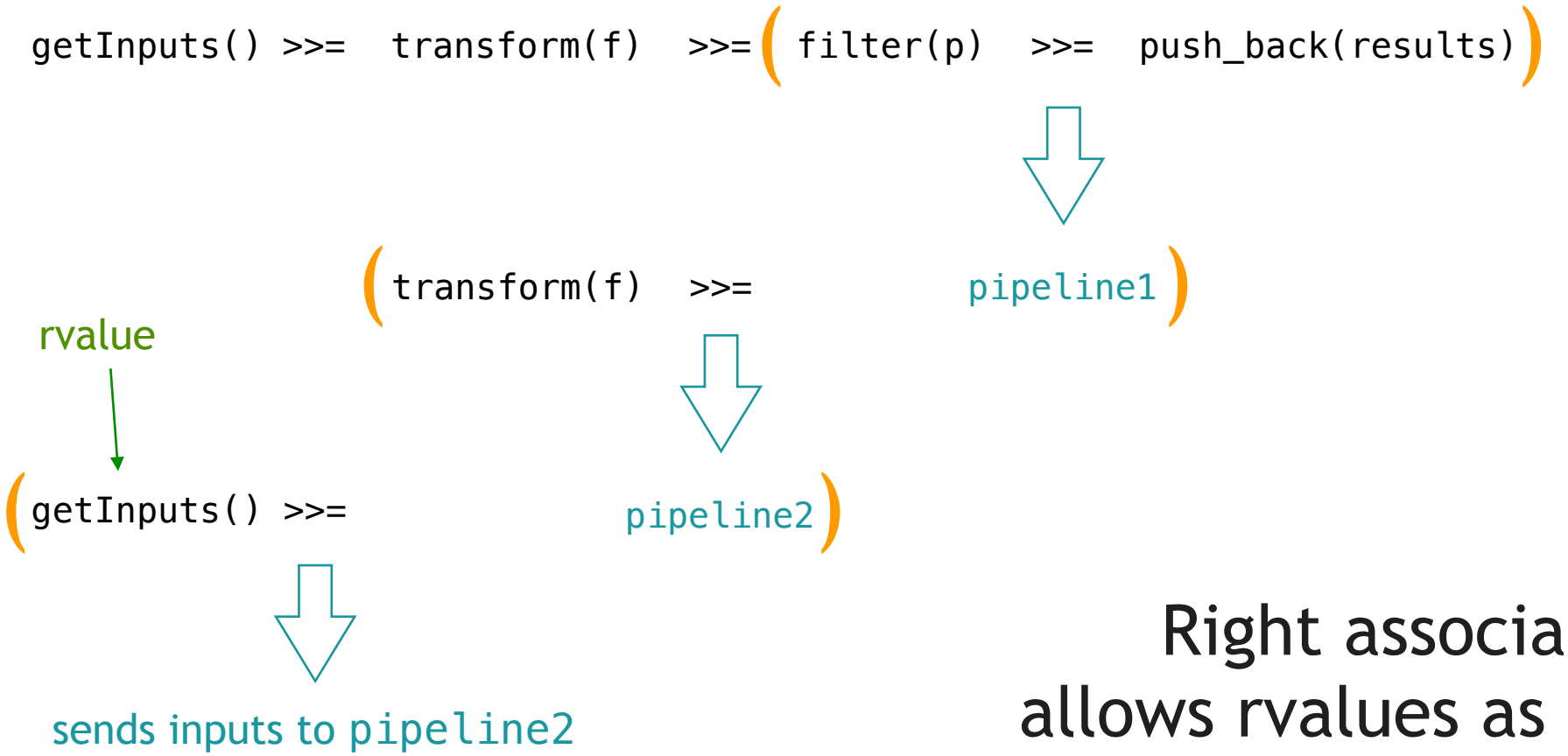


Also, because it is right associative.

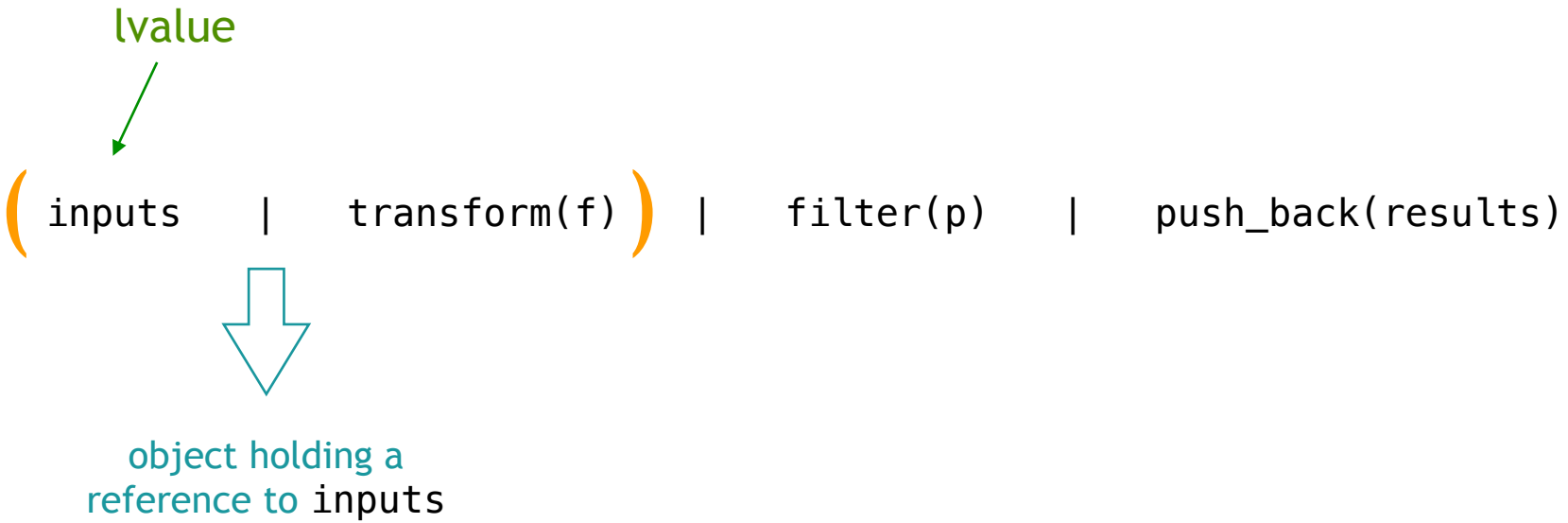
Also, because it is right associative.

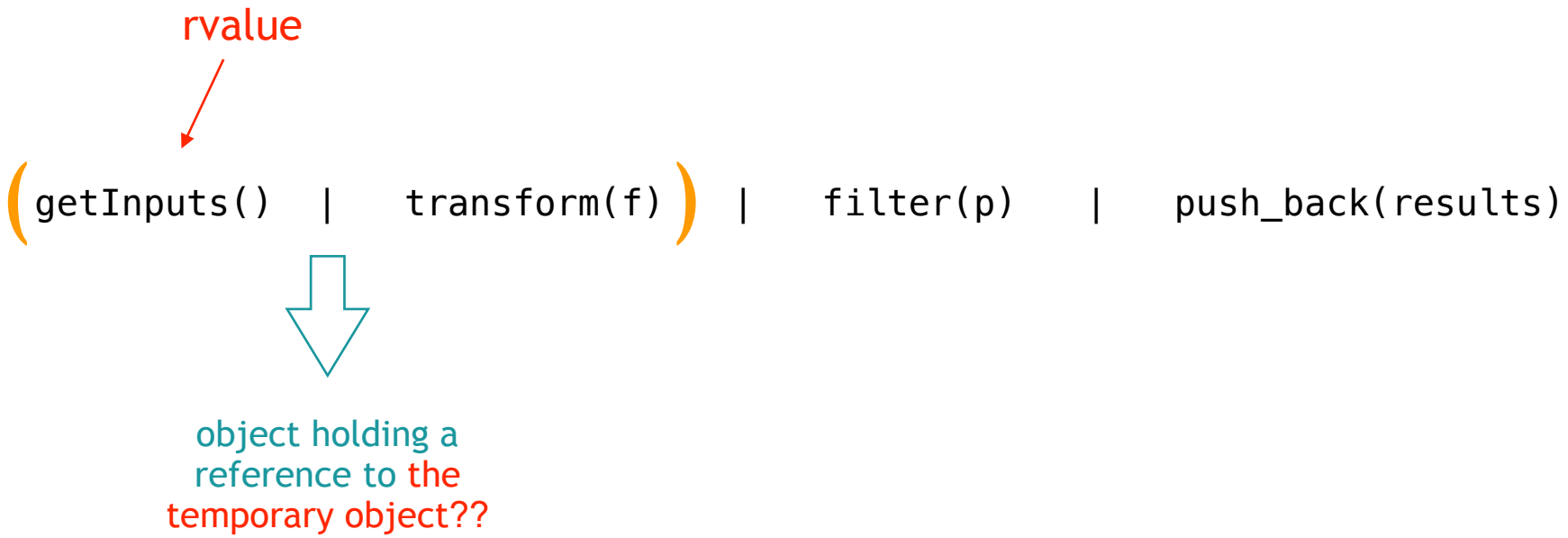


Also, because it is right associative.



Right associativity
allows rvalues as input

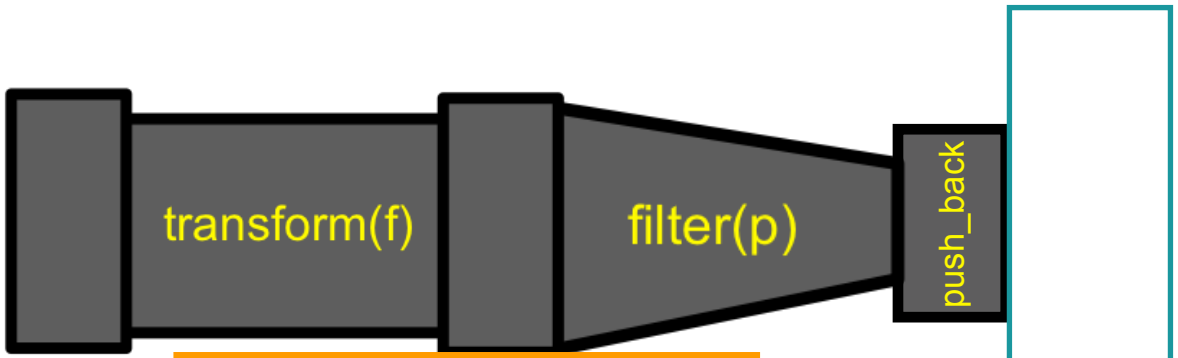

lvalue
(inputs | transform(f)) | filter(p) | push_back(results)
object holding a
reference to inputs




```
std::vector<int> results = getInput()
    | ranges::view::filter([](int i){ return i % 2 == 0; })
    | ranges::view::transform([](int i){ return i * 2; });
```

```
error: static assertion failed: Cannot get a view of a temporary container
 static_assert(std::is_lvalue_reference<T>::value, "Cannot get a view of a temporary container");
```

OUTLINE OF THE REST OF THE TALK



What gets in a pipeline

ranges
STL algorithms
Multiple inputs

The pipes

Inside a pipe
Branching out pipes

The output of a pipeline

Simple container adapters
Complex container adapters
for_each

THE PIPES

pipes::join

pipes::transform

pipes::take

pipes::tap

pipes::take_while

pipes::tee

pipes::filter

pipes::drop

pipes::partition

pipes::drop_while

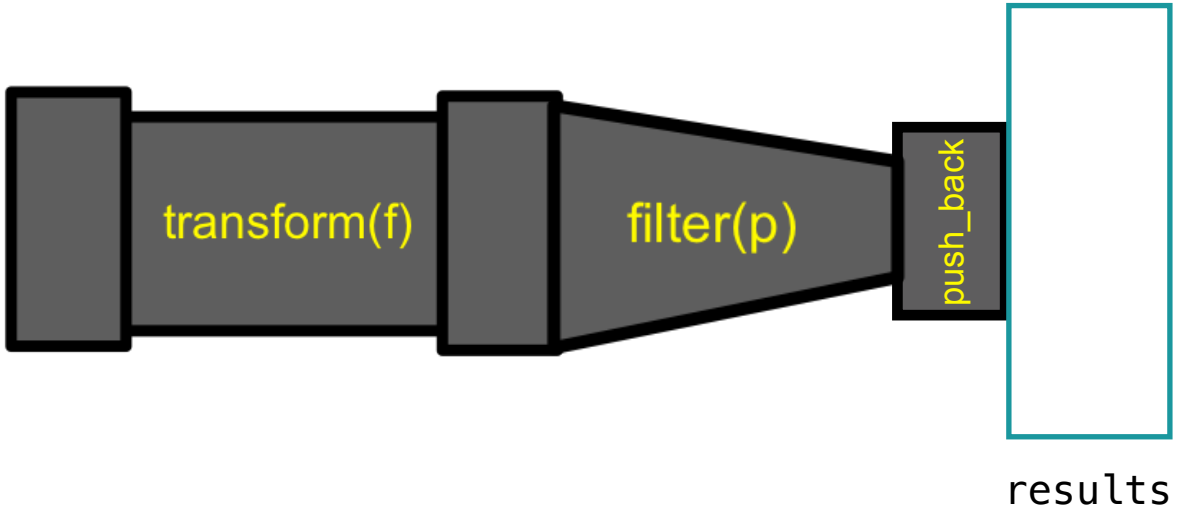
pipes::stride

pipes::intersperse

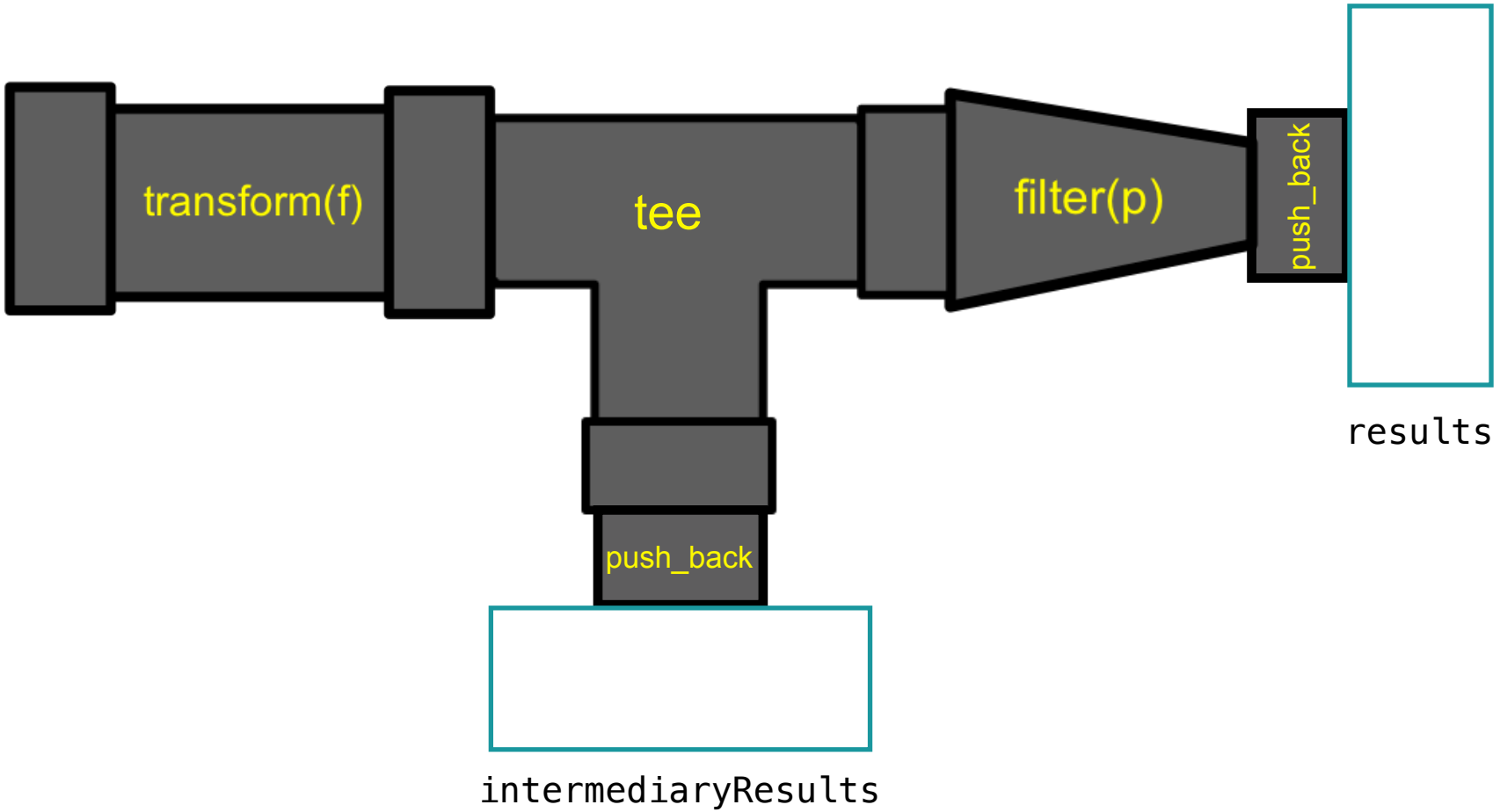
INSIDE A PIPE

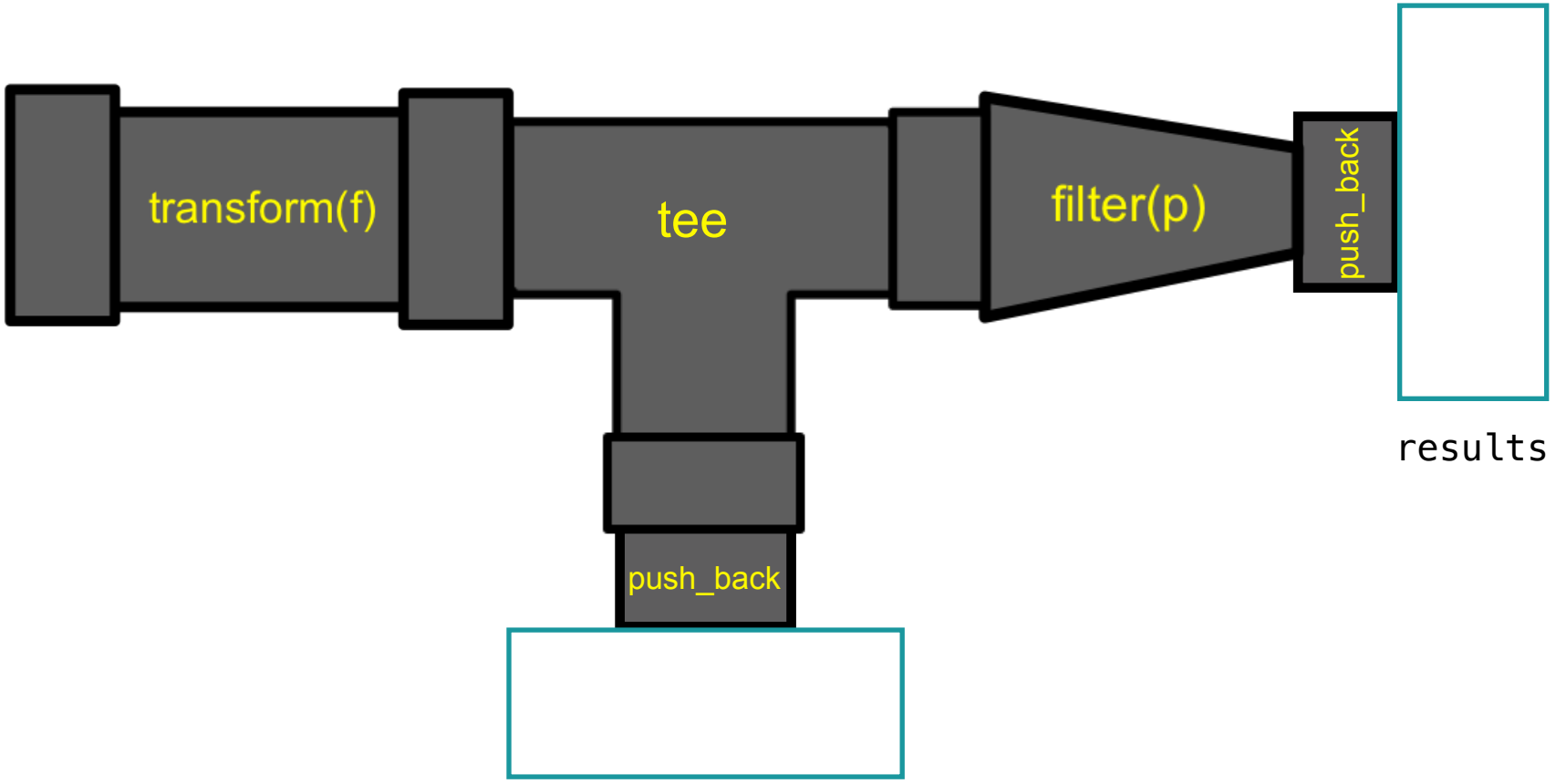
```
pipes::transform
```

BRANCHING OUT

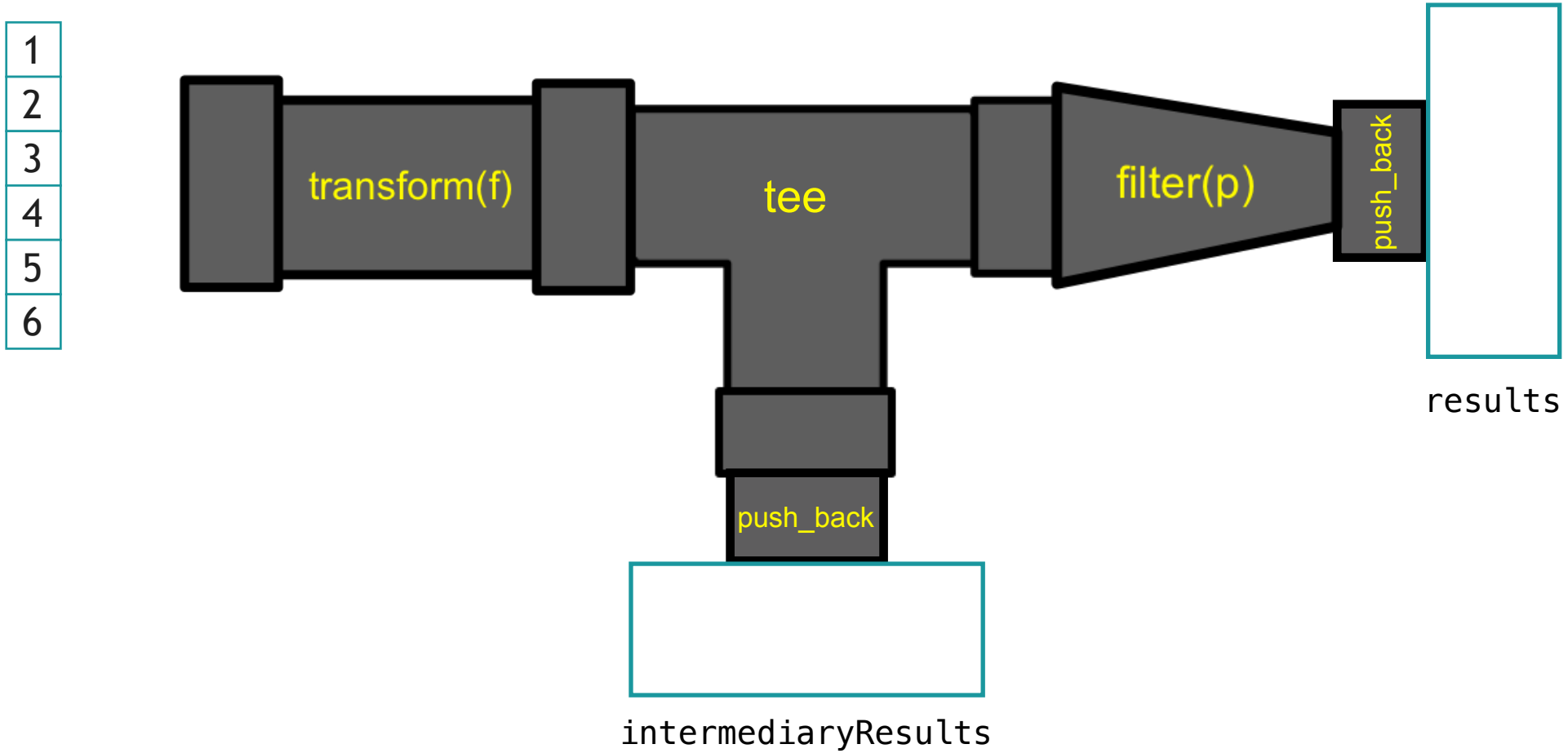


BRANCHING OUT



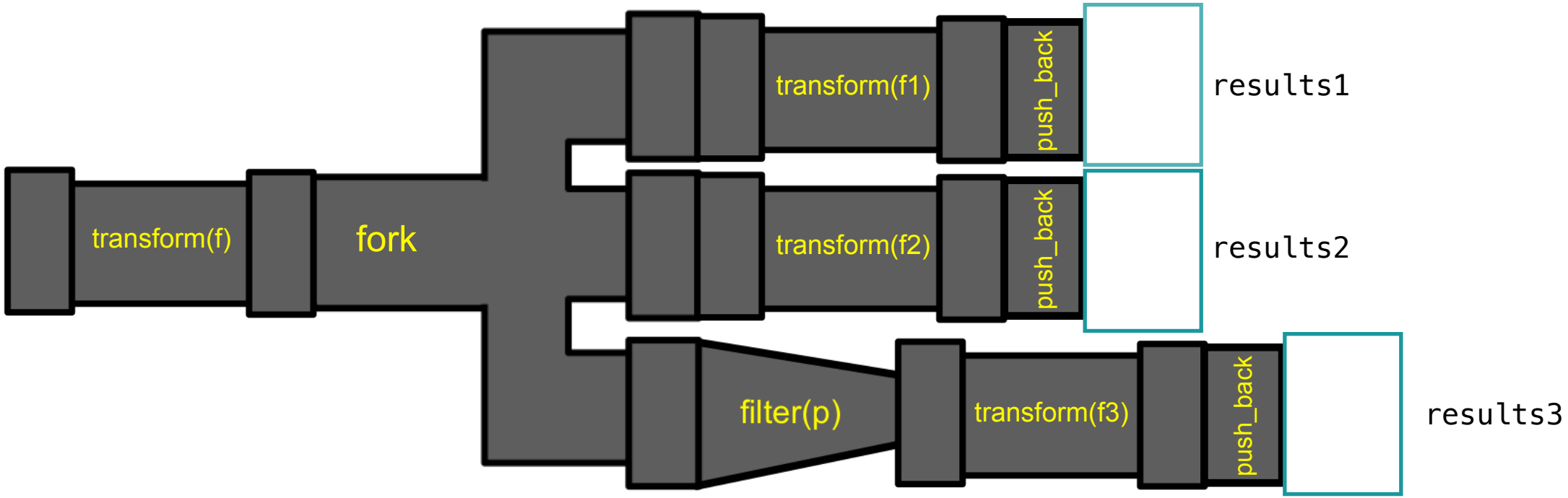


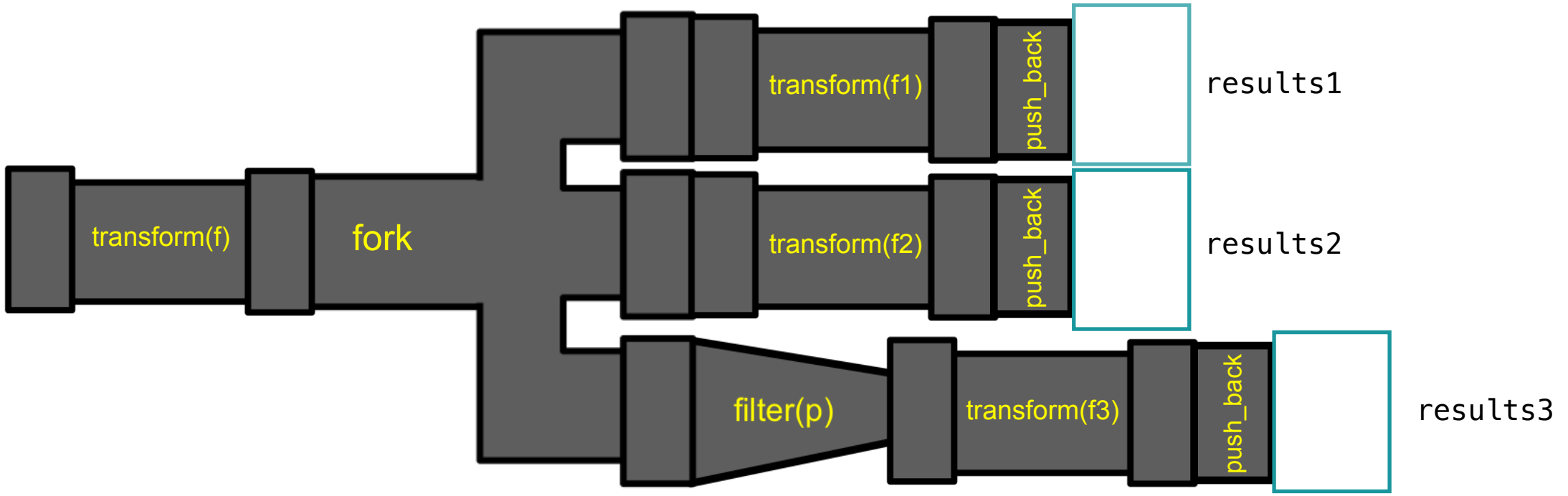

```
auto const inputs = std::vector<int>{1, 2, 3, 4, 5, 6};
auto intermediaryResults = std::vector<int>{};
auto results = std::vector<int>{};

inputs >>= pipes::transform([](int i){ return i * 2; })
       >>= pipes::tee(pipes::push_back(intermediaryResults))
       >>= pipes::filter([](int i){ return i % 3 == 0; })
       >>= pipes::push_back(results);
```




```
auto const inputs = std::vector<int>{1, 2, 3, 4, 5, 6};
auto intermediaryResults = std::vector<int>{};
auto results = std::vector<int>{};

inputs >>= pipes::transform([](int i){ return i * 2; })
       >>= pipes::tee(pipes::push_back(intermediaryResults))
       >>= pipes::filter([](int i){ return i % 3 == 0; })
       >>= pipes::push_back(results);
```

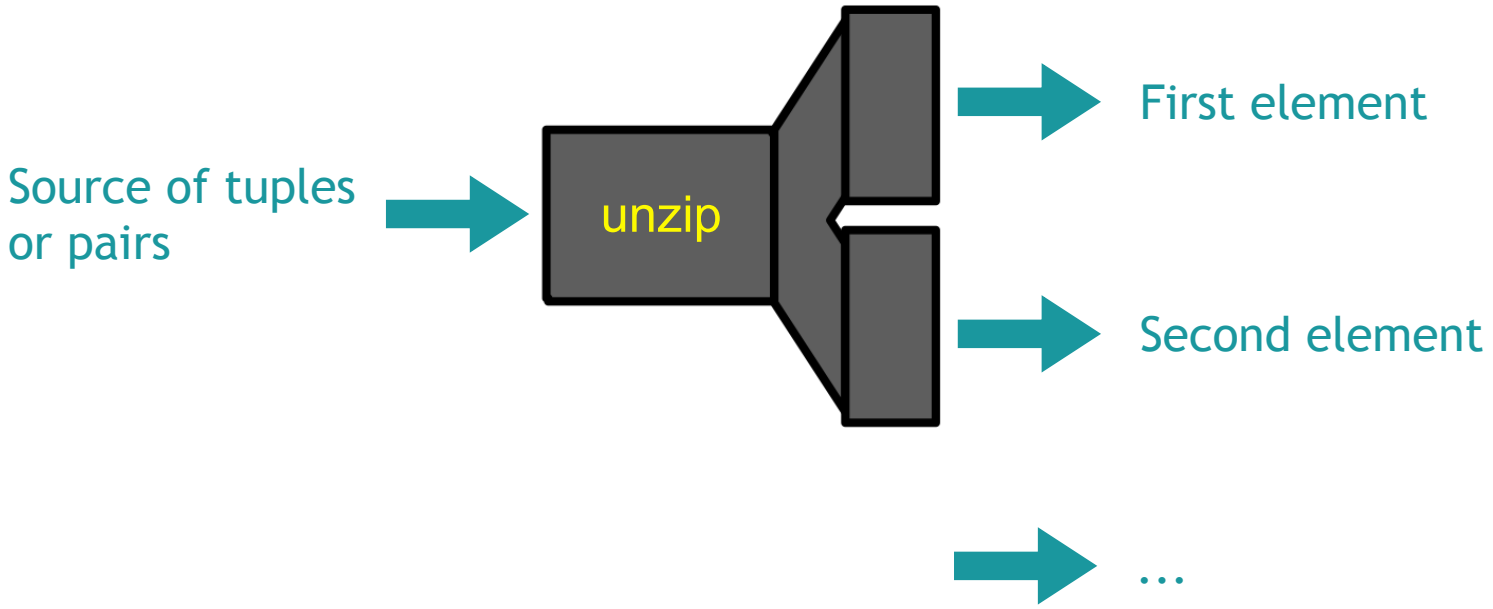
BRANCHING OUT: FORK

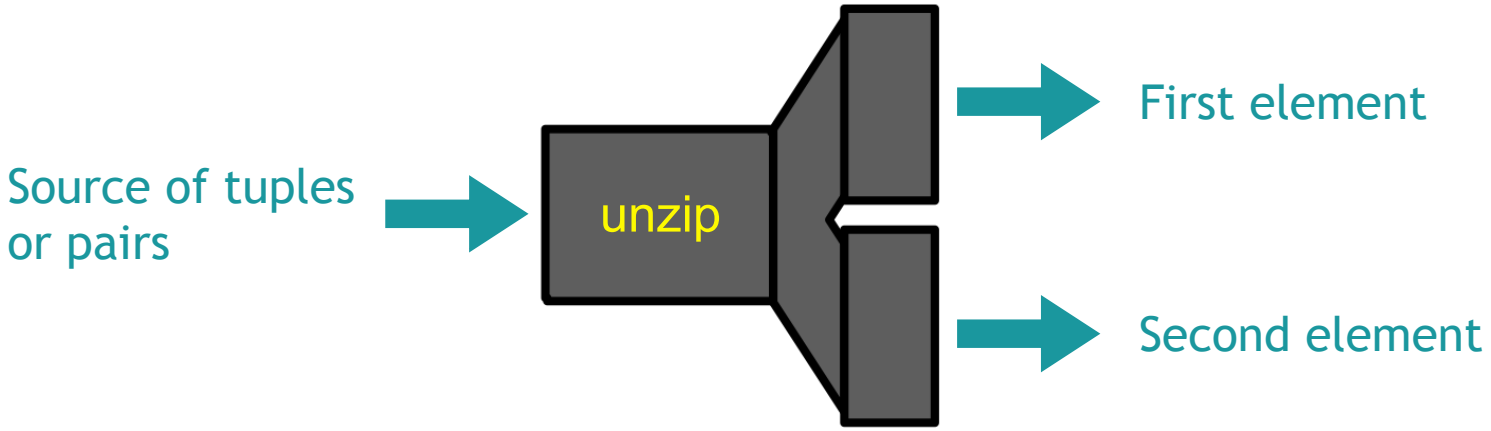





```
pipes::transform(f) >>=
pipes::fork(pipes::transform(f1) >>= pipes::push_back(results1),
            pipes::transform(f2) >>= pipes::push_back(results2),
            pipes::filter(p) >>= pipes::transform(f3) >>= pipes::push_back(results3)));
```

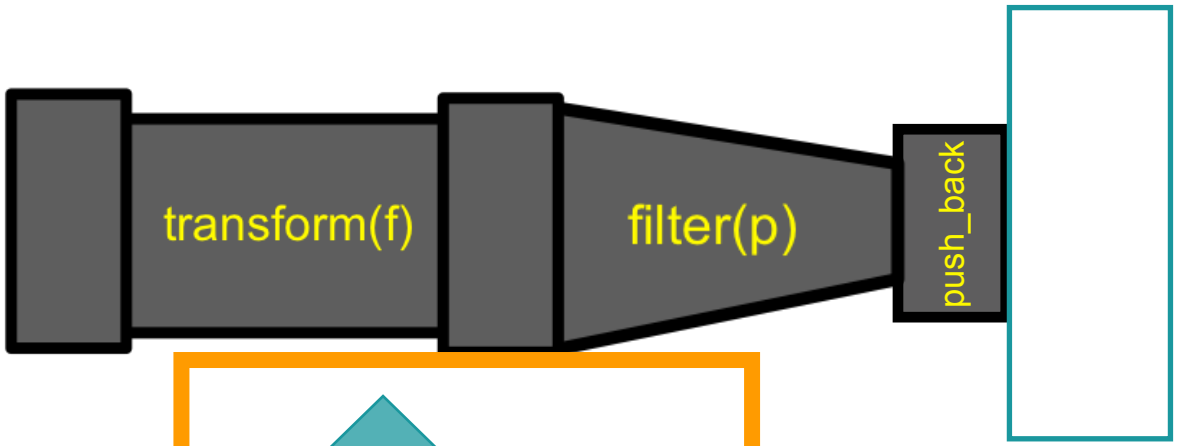
BRANCHING OUT: UNZIP






```
std::map<int, std::string> entries =
    { {1, "one"}, {2, "two"}, {3, "three"}, {4, "four"}, {5, "five"} };

std::vector<int> keys;
std::vector<std::string> values;

entries >>= pipes::unzip(pipes::push_back(keys),
                         pipes::push_back(values)));


entries >>= getKeys(keys);
```

OUTLINE OF THE REST OF THE TALK



What gets in a pipeline

ranges
STL algorithms
Multiple inputs

The pipes

Inside a pipe
Branching out pipes

The output of a pipeline

Simple container adapters
Complex container adapters
for_each

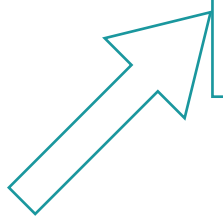
INPUT: RANGES

```
auto const inputs = std::vector<int>{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
auto results = std::vector<int>{};

inputs >>= pipes::transform([](int i){ return i * 2; })
       >>= pipes::filter([](int i){ return i % 3 == 0; })
       >>= pipes::push_back(results);
```

STL range container

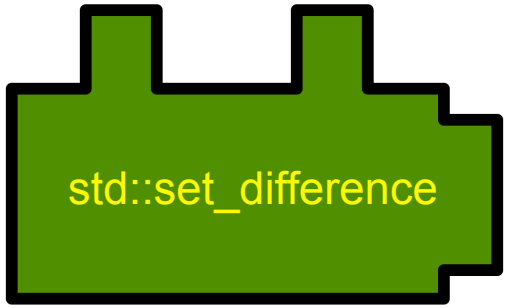
INPUT: RANGES

```
auto const inputs = std::vector<int>{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
auto results = std::vector<int>{};

inputs
    | ranges::view::drop(3)
    | ranges::view::reverse
    >>= pipes::transform([](int i){ return i * 2; })
    >>= pipes::filter([](int i){ return i % 3 == 0; })
    >>= pipes::push_back(results);
```

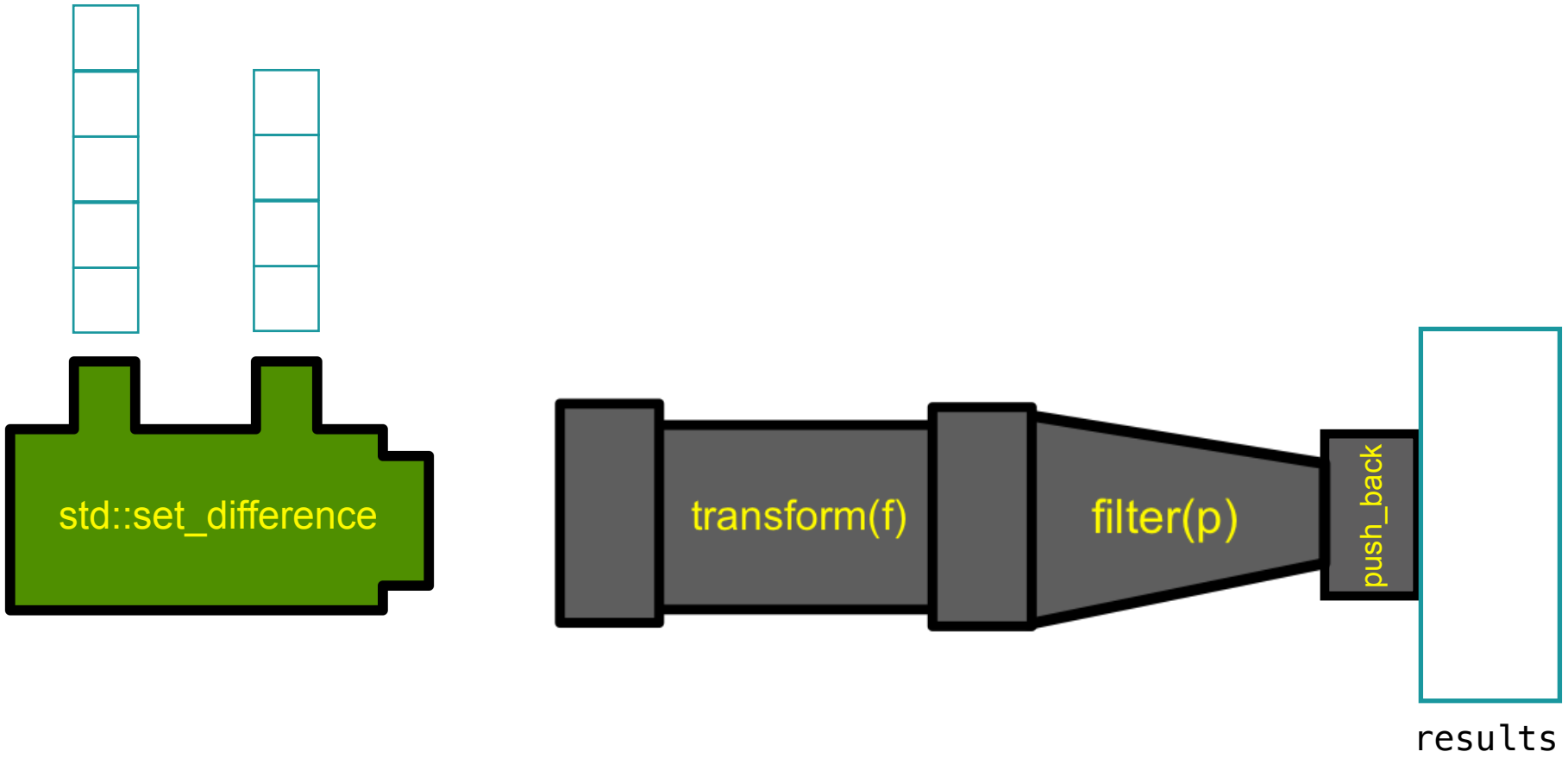


range

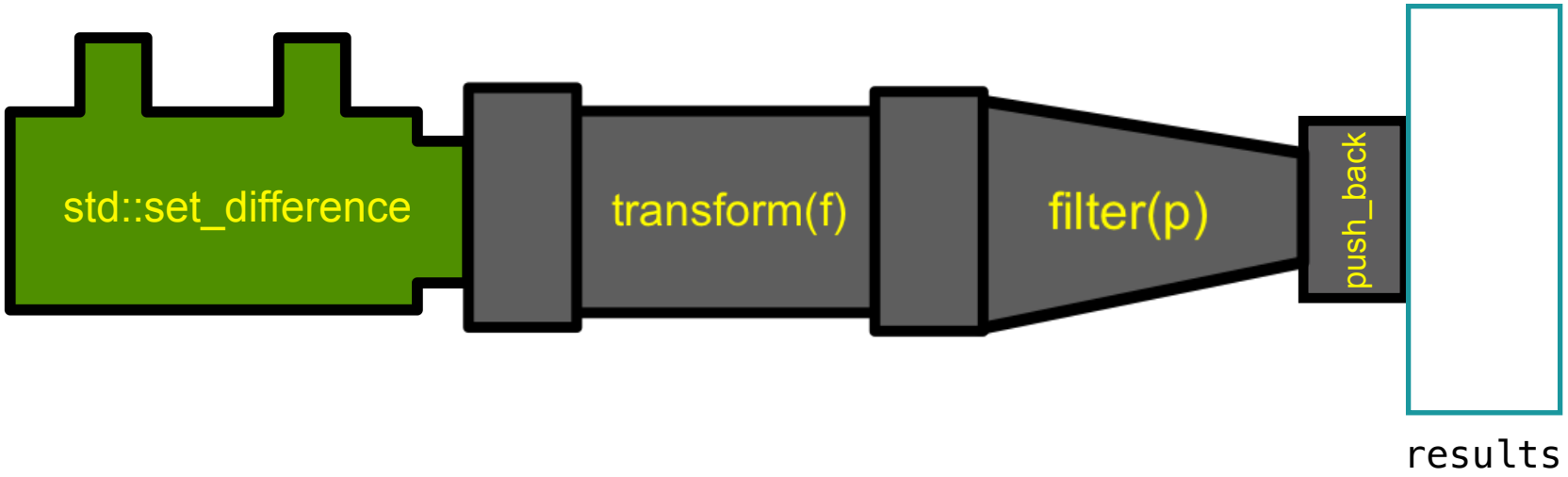
INPUT: STL ALGORITHMS



INPUT: STL ALGORITHMS

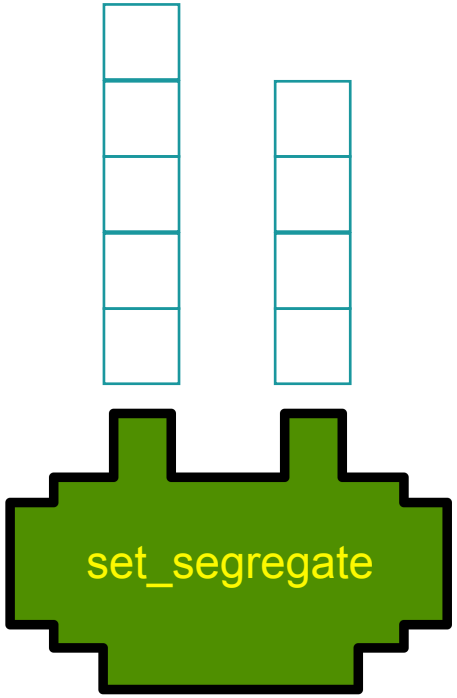


INPUT: STL ALGORITHMS

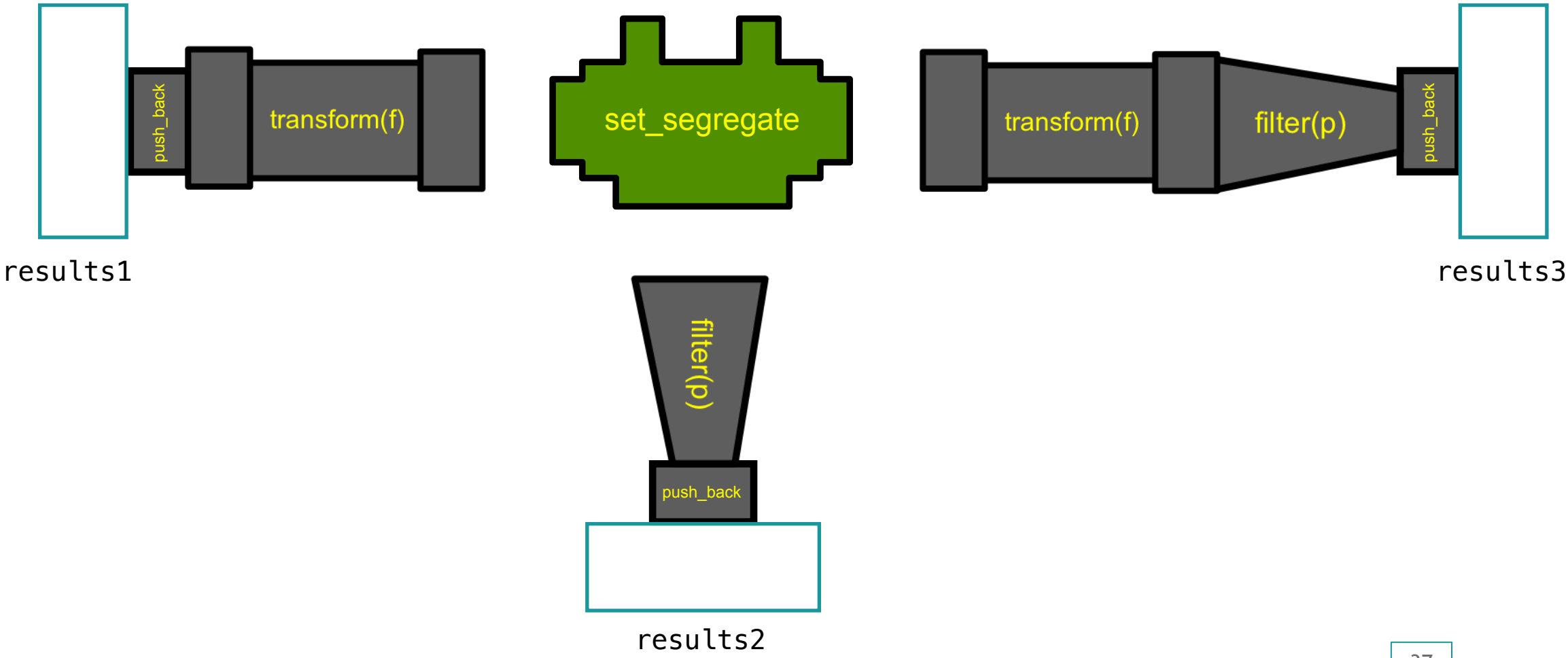



```
std::set_difference(begin(input1), end(input1),
                    begin(input2), end(input2),
                    pipes::transform([](int i){ return i * 2; })
                 >>= pipes::filter([](int i){ return i % 3 == 0; })
                 >>= pipes::push_back(results));
```

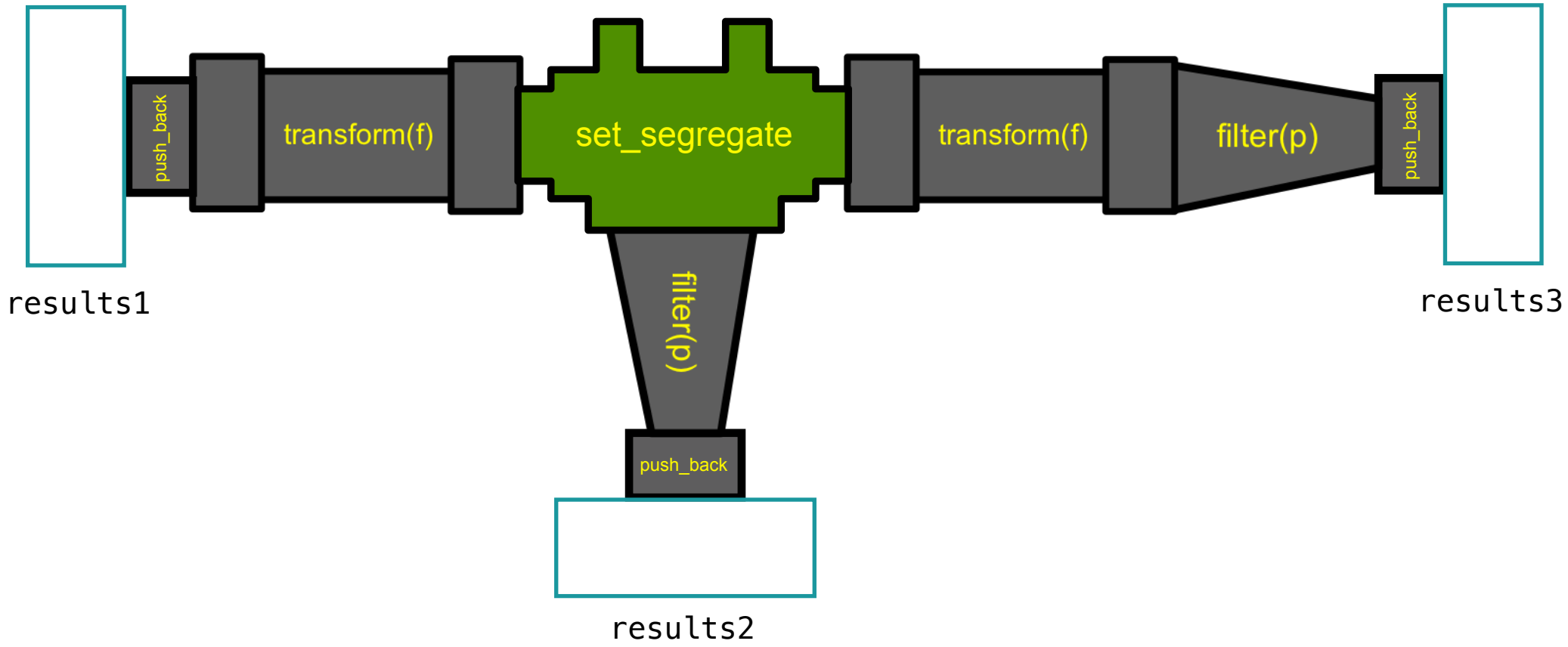
INPUT: STL ALGORITHMS



INPUT: STL ALGORITHMS



INPUT: STL ALGORITHMS

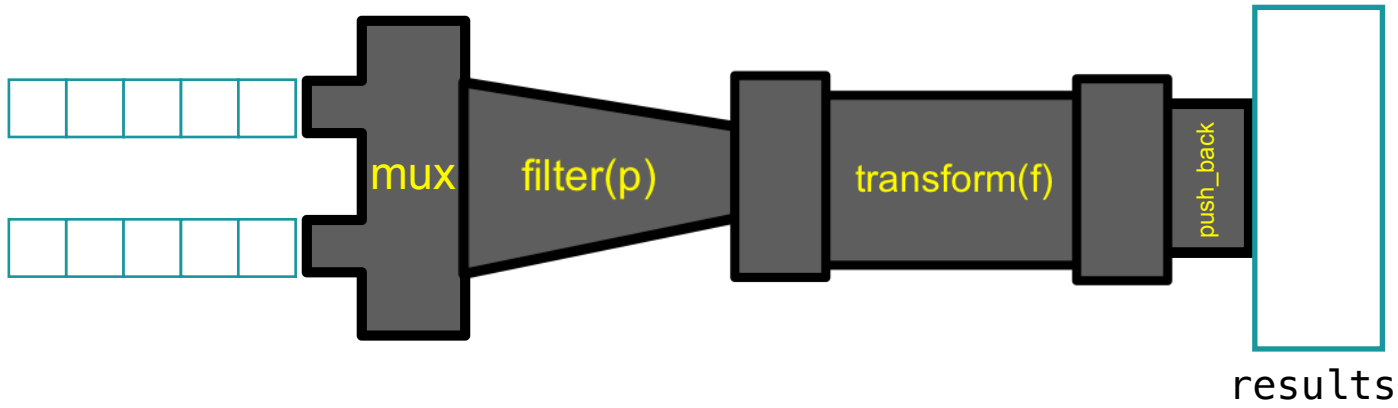



```
set_segregate(input1, input2,
              pipes::transform(f1) >>= pipes::push_back(results1),
              pipes::filter(p2) >>= pipes::push_back(results2),
              pipes::transform(f3) >>= pipes::filter(p3) >>= pipes::push_back(results));
```

INPUT: SEVERAL RANGES

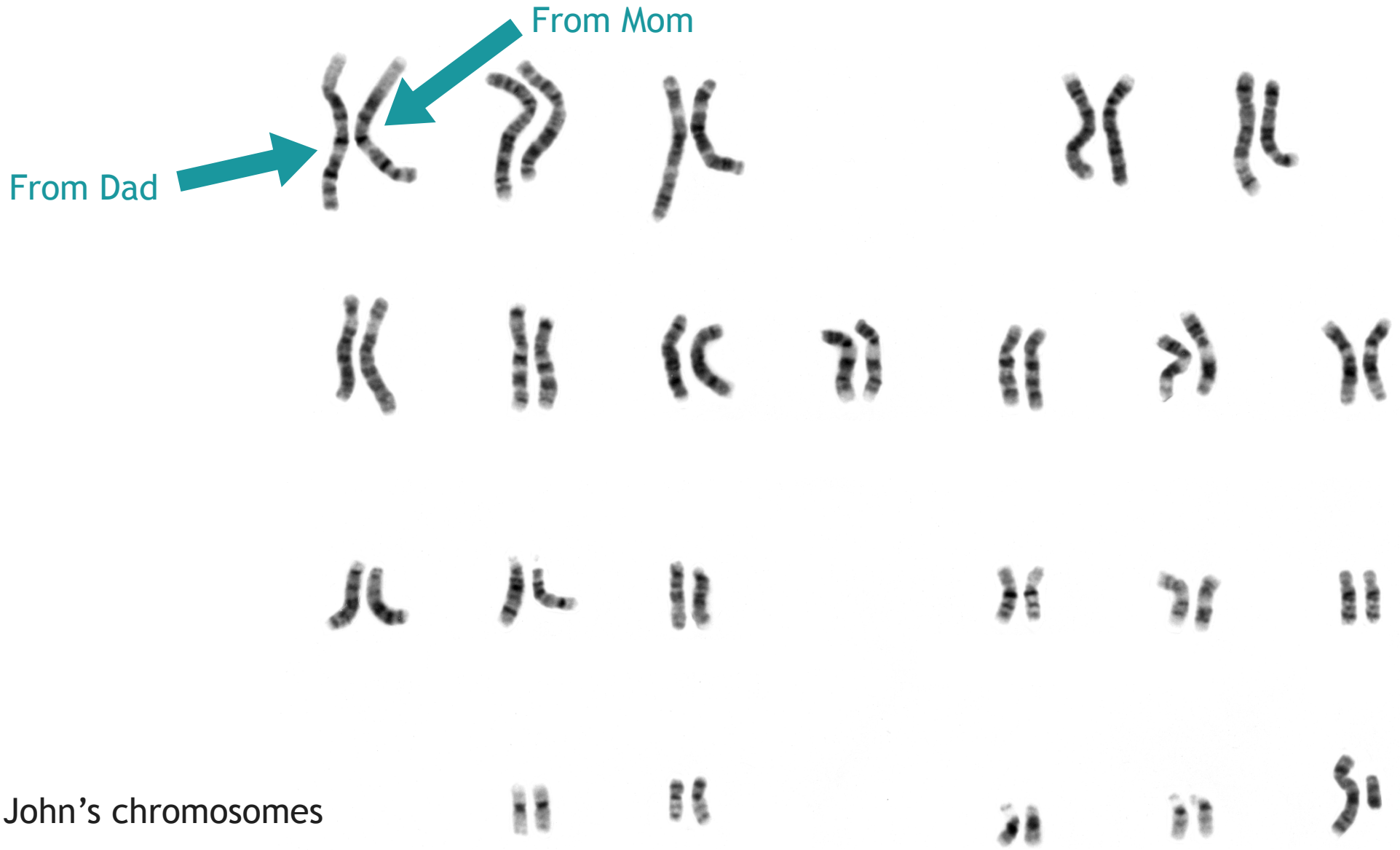
```
auto const inputs1 = std::vector<int>{1, 2, 3, 4, 5};
auto const inputs2 = std::set<int>{10, 20, 30, 40, 50};

std::vector<int> results =
    ranges::view::zip(inputs1, inputs2)
  | ranges::view::filter([](auto&& values){ auto const& [a,b] = values; return a + b < 41; })< 41; })t<1>(values) < 41; })
  | ranges::view::transform([](auto&& values){ auto const& [a,b] = values; return a * b; });ıes); });::get<1>(values); });
```

INPUT: SEVERAL RANGES




```
auto const input1 = std::vector<int>{1, 2, 3, 4, 5};
auto const input2 = std::vector<int>{10, 20, 30, 40, 50};
auto results = std::vector<int>{};

pipes::mux(input1, input2)
    >>= pipes::filter([](int a, int b){ return a + b < 41; })
    >>= pipes::transform(std::multiplies{})
    >>= pipes::push_back(results);
```

No tuple

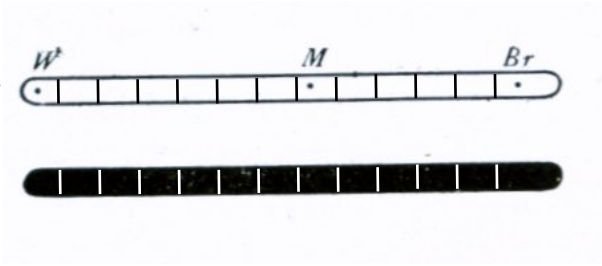


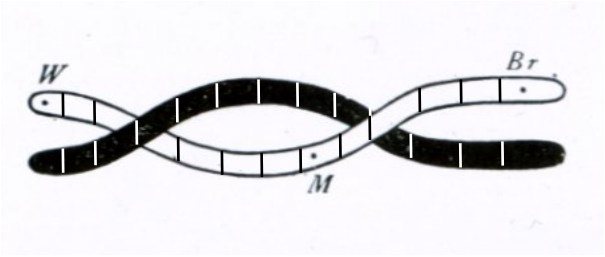
John’s chromosomes

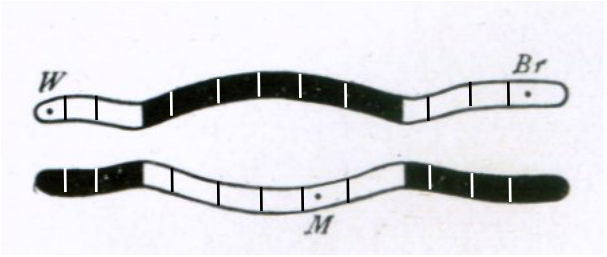
THE CROSSING OVER

From John's Dad

From John's Mom





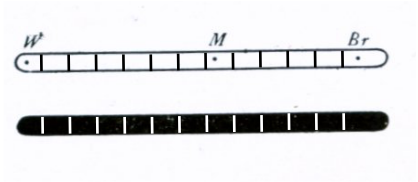


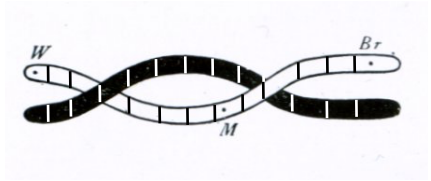
To John's kids

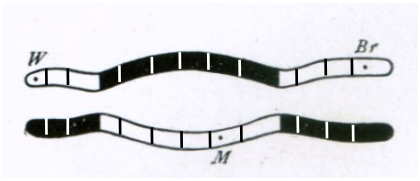
To John's kids

THE CROSSING OVER

From John's Dad →
From John's Mom →

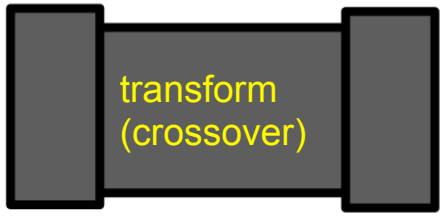


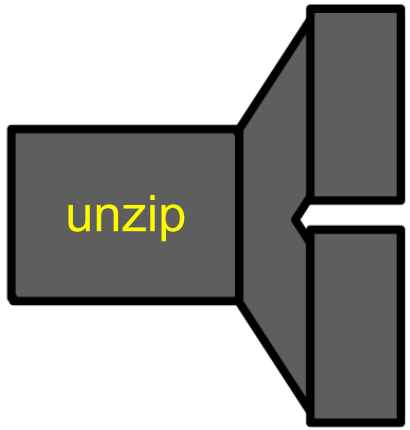




→ To John's kids
→ To John's kids

From John's Dad →
From John's Mom →

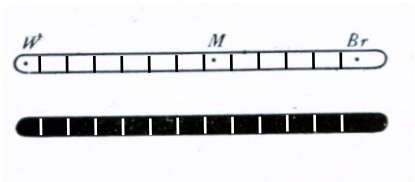


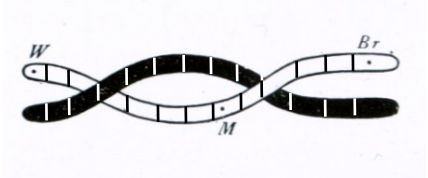


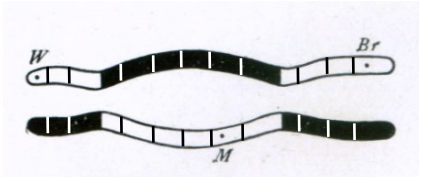
→ To John's kids
→ To John's kids

From John's Dad

From John's Mom





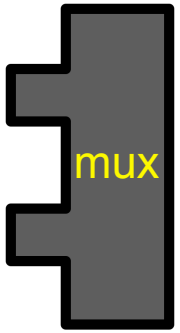


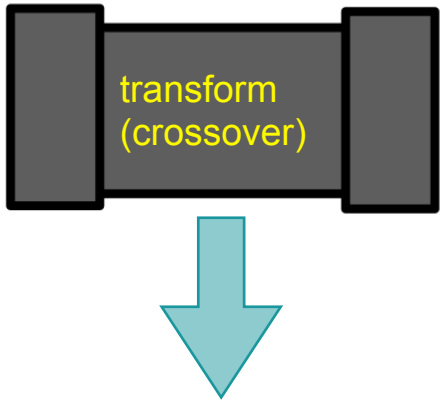
To John's kids

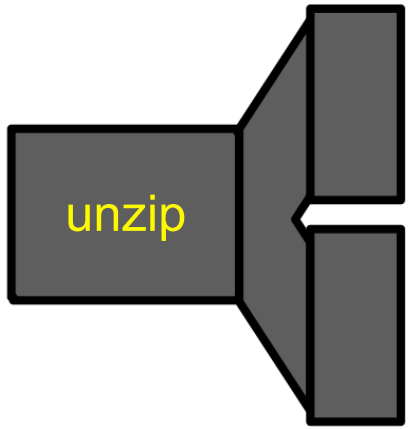
To John's kids

From John's Dad

From John's Mom



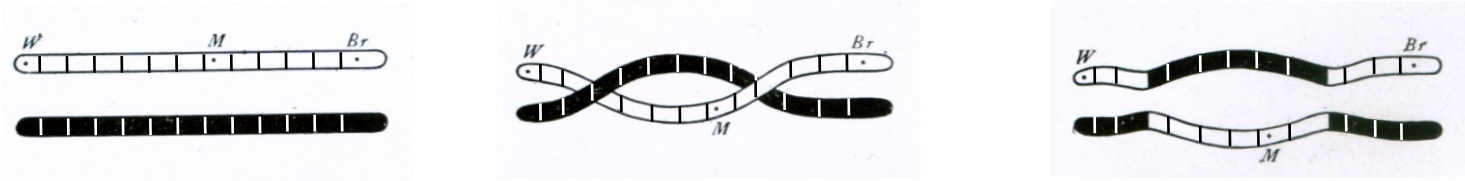


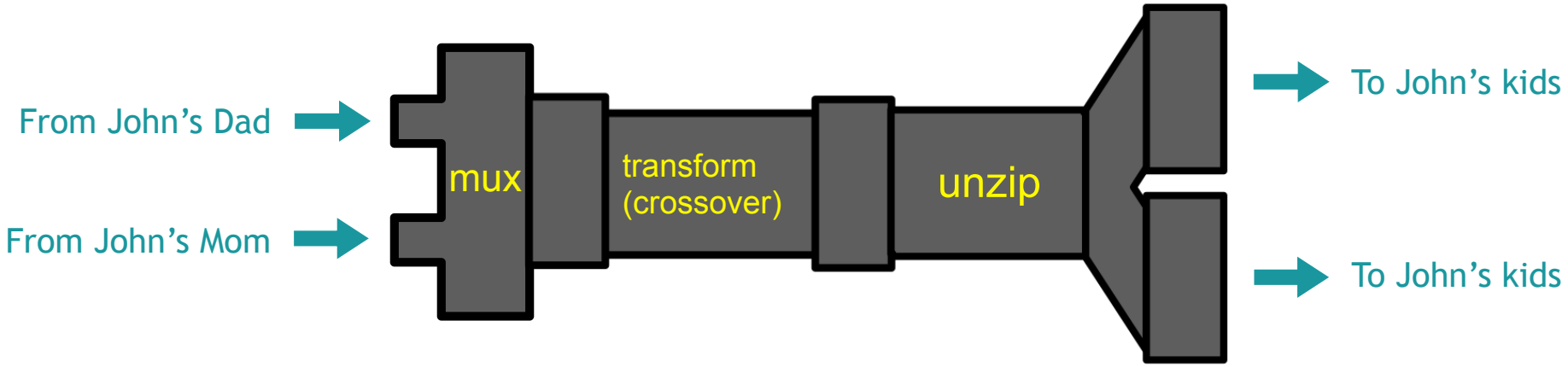


To John's kids

To John's kids

```
std::pair<char, char> crossover(char dadGene, char momGene)
{
    static auto generateRandomNumber = RandomNumberGenerator{0, 1};

    if (generateRandomNumber() == 1)
    {
        return { dadGene, momGene };
    }
    else
    {
        return { momGene, dadGene };
    }
}
```






```
auto const dadChromosome = std::string(25, 'd');
auto const momChromosome = std::string(25, 'm');

auto childChromosome1 = std::string{};
auto childChromosome2 = std::string{};

pipes::mux(dadChromosome, momChromosome)
    >>= pipes::transform(crossover)
    >>= pipes::unzip(pipes::push_back(childChromosome1),
                     pipes::push_back(childChromosome2));

std::cout << childChromosome1 << '\n';
std::cout << childChromosome2 << '\n';
```

```
ddmdmdddmmmmmmmdmmmmmddddd
mmdmdmmmddddddddmddddddmmmm
```

```cpp
std::string numberDashColor(int number, std::string const& color)
{
return std::to_string(number) + '-' + color ;
}


auto const numbers = std::vector<int>{ 1, 2, 3 };
auto const colors = std::vector<std::string>{ "blue", "red", "green" };

auto results = std::vector<std::string>{};

pipes::mux(numbers, colors)
    >>= pipes::transform(numberDashColor)
    >>= pipes::intersperse("\n"));
    >>= pipes::to_out_stream(std::cout);
```

```
1-blue
2-red
3-green
```

```cpp
std::string numberDashColor(int number, std::string const& color)
{
return std::to_string(number) + '-' + color ;
}


auto const numbers = std::vector<int>{ 1, 2, 3 };
auto const colors = std::vector<std::string>{ "blue", "red", "green" };



pipes::cartesian_product(numbers, colors)
    >>= pipes::transform(numberDashColor)
    >>= pipes::intersperse("\n")
    >>= pipes::to_out_stream(std::cout);
```

```
1-blue
1-red
1-green
2-blue
2-red
2-green
3-blue
3-red
3-green
```

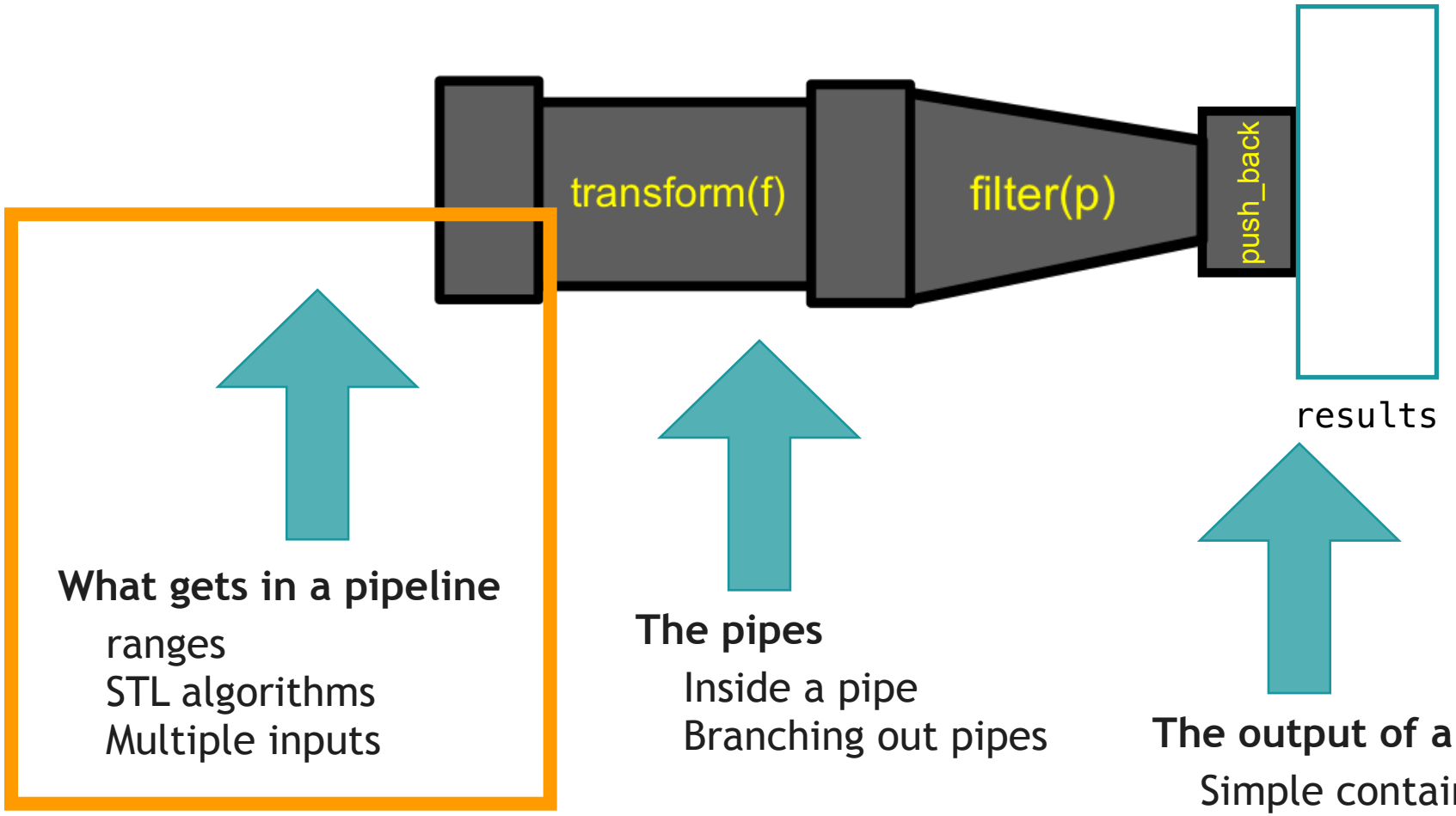
```cpp
auto const numbers = std::vector<int>{ 1, 2, 3, 4, 5 };

pipes::combinations(numbers)
    >>= pipes::transform(numberDashNumber)
    >>= pipes::intersperse("\n")
    >>= pipes::to_out_stream(std::cout);
```

```
1 - 2
1 - 3
1 - 4
1 - 5
2 - 3
2 - 4
2 - 5
3 - 4
3 - 5
4 - 5
```

```cpp
auto const inputs = std::vector<std::string>{ "one", "two", "three", "four" };

pipes::adjacent(inputs)
    >>= pipes::transform(valueFollowsValue)
    >>= pipes::intersperse("\n")
    >>= pipes::to_out_stream(std::cout);
```

```
two follows one
three follows two
four follows three
```

OUTLINE OF THE REST OF THE TALK



What gets in a pipeline

ranges
STL algorithms
Multiple inputs

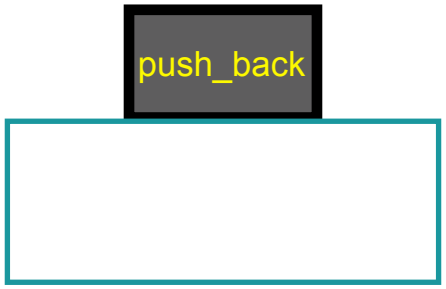
The pipes

Inside a pipe
Branching out pipes

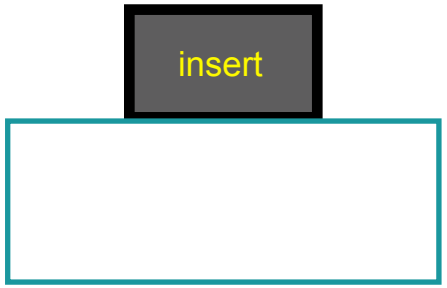
The output of a pipeline

Simple container adapters
Complex container adapters
for_each

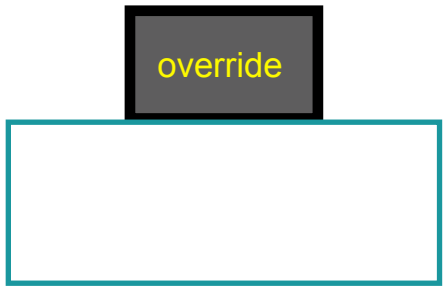
STL equivalent



`std::back_inserter`

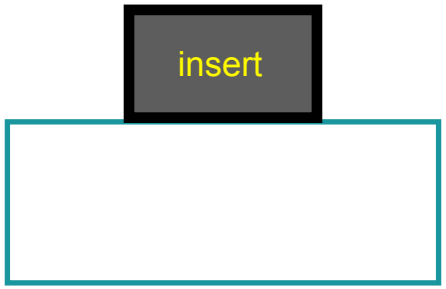


`std::inserter`



`std::begin`

STL equivalent



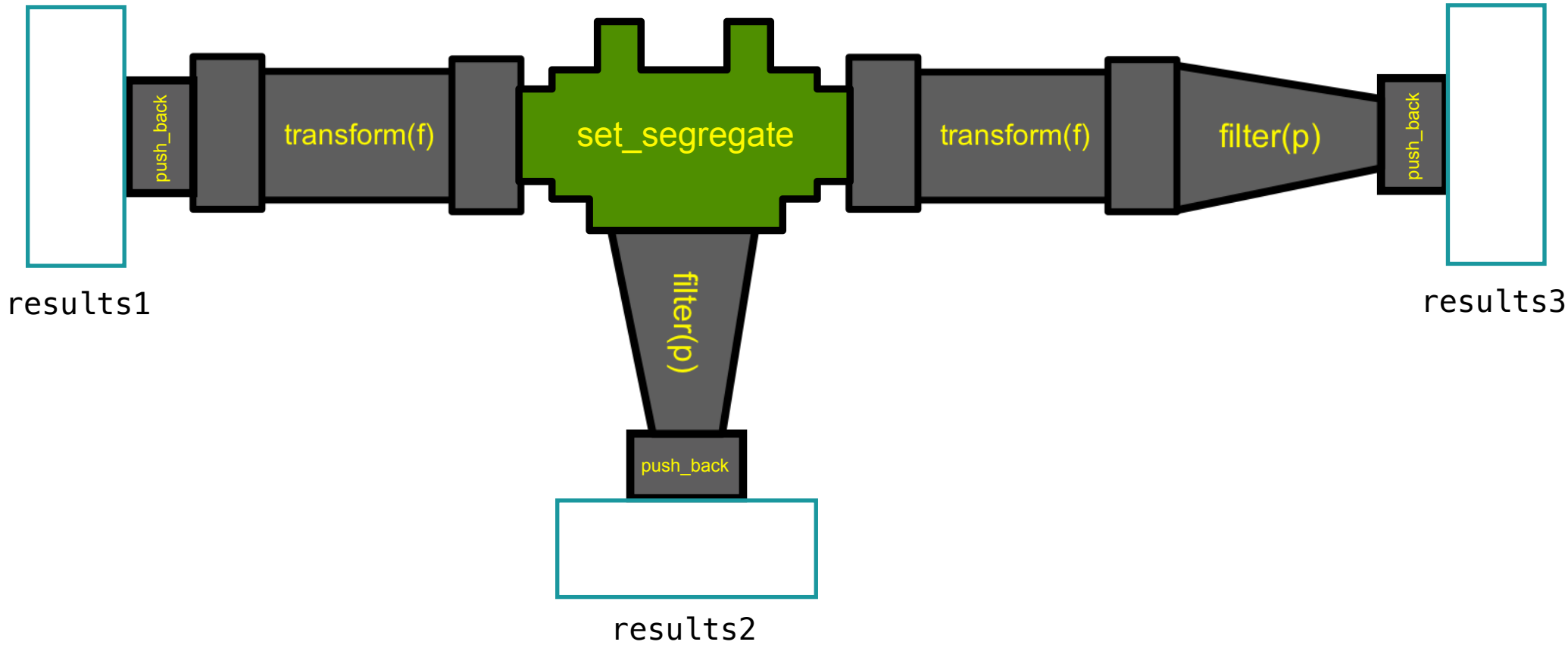
`std::inserter` (kind of)

```
auto input = std::vector<int>{3, 2, 1, 2, 3};
auto results = std::set<int>{};

input >>= pipes::transform([](int i){ return i * 2; })
      >>= pipes::insert(results);lts, end(results)),
                [](int i){ return i * 2; });
```

???

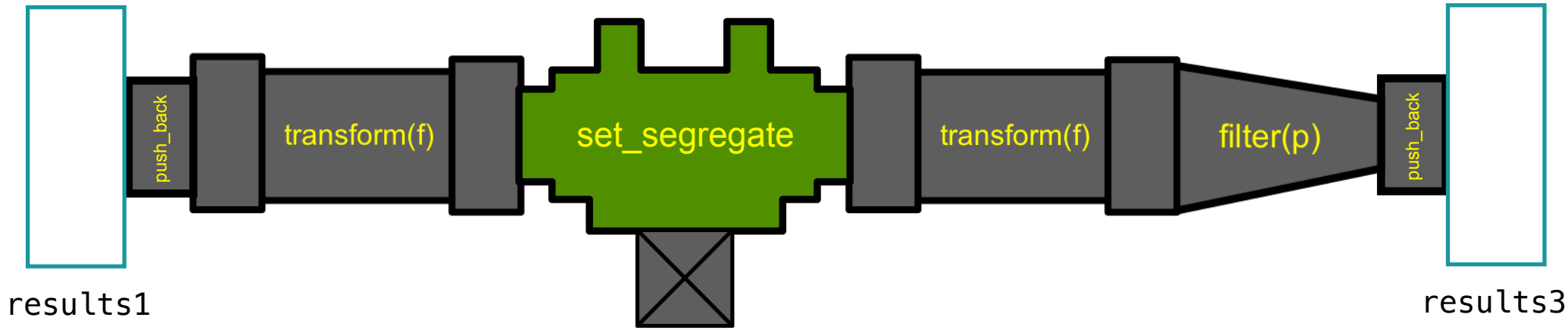
dev_null

INPUT: STL ALGORITHMS




```
set_segregate(input1, input2,
              pipes::transform(f1) >>= pipes::push_back(results1),
              pipes::filter(p2) >>= pipes::push_back(results2),
              pipes::transform(f3) >>= pipes::filter(p3) >>= pipes::push_back(results));
```

INPUT: STL ALGORITHMS




```
set_segregate(input1, input2,
              pipes::transform(f1) >>= pipes::push_back(results1),
              pipes::dev_null,
              pipes::transform(f3) >>= pipes::filter(p3) >>= pipes::push_back(results));
```

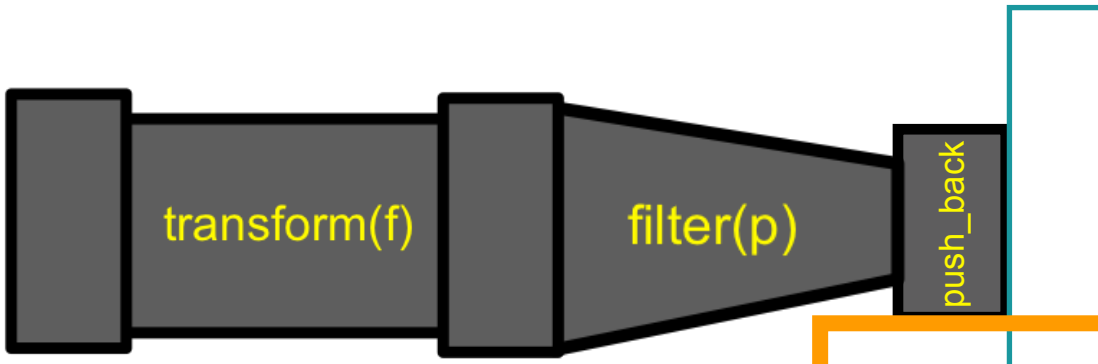
CUSTOM END PIPE

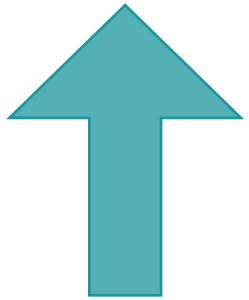
```
auto const inputs = std::vector<int>{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
auto results = std::vector<int>{};

inputs >>= pipes::transform([](int i){ return i * 2; })
       >>= pipes::filter([](int i){ return i % 3 == 0; })
       >>= pipes::for_each(doMyCustomTreatment);
```

INTEGRATION IN MAP

```
std::vector<std::pair<int, std::string>> myMap = { {1, "a"}, {2, "b1"}, {3, "c1"} };
std::vector<std::pair<int, std::string>> newEntries = { {2, "b2"}, {3, "c2"}, {4, "d"} };

newEntries >>= map_aggregator(myMap, std::plus{});rings);

// mayMap contains { {1, "a"}, {2, "b1b2"}, {3, "c1c2"}, {4, "d"} };

std::string concatenateStrings(std::string const& s1, std::string const& s2)
{
    return s1 + s2;
}
```

OUTLINE OF THE REST OF THE TALK



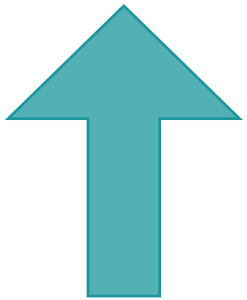


What gets in a pipeline

ranges
STL algorithms
Multiple inputs

The pipes

Inside a pipe
Branching out pipes



The output of a pipeline

Simple container adapters
Complex container adapters
for_each

LIMITATIONS

Pipes don’t know what’s coming next. How to implement `reverse`? `drop_last`?

Pipes don’t cache input data. How to implement `split`?

Can’t send a range to an individual pipe: `inputs >>= transform(f);`

Have to create the results container on a separate statement

STRENGTHS

`transform` called only once per element

rvalues accepted as inputs

Branching out: `tee`, `demux`, `unzip`

Pick up results from STL algorithms

Multiple inputs without tuples

Expressive insertion in containers: `map_aggregator`, etc.

Easy to implement

CONTRIBUTIONS

github.com/JoBoccara/pipes

- `input >>= transform(f) >>= push_back(r);`
  `input >>= transform(f) >>= insert(r);`
- `input >>= transform(f) >>= r;`

What's a cool name for >>= ??

Fluent{C++}

fluentcpp.com