C++ now

Rappel: Compose Algorithms, Not Iterators

Google's Alternative to Ranges

John Bandela & Chris Philip

2024

HISTORY

by Jon Bentley
with Special Guest Oysters
Don Knuth and Doug McIlroy

programming pearls

A LITERATE PROGRAM

1986

CHALLENGE

Given a text file and an integer k, print the k most common words in the file (and the number of their occurrences) in decreasing frequency.

DON KNUTH

Produces an absolute virtuoso work of art using literate programming and WEB.

- Elegant Pascal code
- Tries
- Beautifully typeset with cross-references

DOUG MCILROY

```
tr -cs A-Za-z`
` |
tr A-Z a-z |
sort |
uniq -c |
sort -rn |
sed ${1} q
```

PIPELINE STYLE

- Writing a series of transformations as a single expression without nested parentheses, rather than the more standard style of using separate invocations.
- Common in other languages
  - Bash uses | to “pipe” data through a sequence of programs.
  - F# uses |>
- The primary syntactic feature is the lack of unbounded nesting.

C++ PSEUDOCODE

```
// Standard C++, non-pipelined
auto result1 = f1(input);
auto result2 = f2(result1);
auto result3 = f3(result2);

// Pipeline style
auto result = input | f1 | f2 | f3;
```

TERMINOLOGY

- Pipeline
  - The entire series
  - `input | f1 | f2 | f3;`
- Stage
  - Each individual component of the pipeline
    - f1
    - f2
    - f3
  - Also have seen *combinator* used for this, but this talk will use *stage*

STD::RANGES

```
auto const ints = {0, 1, 2, 3, 4, 5};
auto even = [](int i) { return 0 == i % 2; };
auto square = [](int i) { return i * i; };

for (int i : ints
             | std::views::filter(even)
             | std::views::transform(square))
    std::cout << i << ' ';
```

- https://en.cppreference.com/w/cpp/ranges#Range_ad

A MORE COMPLICATED EXAMPLE

```
using IntAndString = std::pair<int, std::string>;
auto make_int_and_string = [](int i) -> IntAndString {
    return {i*i*i, std::to_string(i)};
};
auto result = std::views::iota(1,1000001)
  | std::views::transform(make_int_and_string)
  | std::views::filter([](const auto& p) {
       return p.first >= std::hash<std::string>()(p.second);
   })
  | std::views::transform(&IntAndString::second)
  | std::views::take(4);
for (auto s : result)
  std::cout << s << "\n";
```

CRASH

WHY?

```
using IntAndString = std::pair<int, std::string>;
auto make_int_and_string = [](int i) -> IntAndString {
    return {i*i*i, std::to_string(i)};
};
auto result = std::views::iota(1,1000001)
  | std::views::transform(make_int_and_string)
  | std::views::filter([](const auto& p) {
       return p.first >= std::hash<std::string>()(p.second);
   })
  | std::views::transform(&IntAndString::second)
  | std::views::take(4);
for (auto s : result)
  std::cout << s << "\n";
```

WHY?

```
using IntAndString = std::pair<int, std::string>;
auto make_int_and_string = [](int i) -> IntAndString {
    return {i*i*i, std::to_string(i)};
};
auto result = std::views::iota(1,1000001)
  | std::views::transform(make_int_and_string)
  | std::views::filter([](const auto& p) {
       return p.first >= std::hash<std::string>()(p.second);
   })
  | std::views::transform(&IntAndString::second)
  | std::views::take(4);
for (auto s : result)
  std::cout << s << "\n";
```

REDUCED

```
using IntAndString = std::pair<int, std::string>;
auto make_int_and_string = [](int i) -> IntAndString {
    return {i*i*i, std::to_string(i)};
};
auto result = std::views::iota(1,1000001)
  | std::views::transform(make_int_and_string)
  | std::views::transform(&IntAndString::second)
for (auto s : result)
  std::cout << s << "\n";
```

TRANSFORM ITERATORS

```
struct MakeIntAndStringIterator{
  IotaIterator iota_iter;
  IntAndString operator*(){
    return make_int_and_string(*iota_iter);
  }
};

struct SecondIterator{
  MakeIntAndStringIterator int_and_string_iter;
  std::string& operator*(){
    return (*int_and_string_iter).second;
  }
}
```

TRANSFORM ITERATORS

```
struct MakeIntAndStringIterator{
  IotaIterator iota_iter;
  IntAndString operator*(){
    return make_int_and_string(*iota_iter);
  }
};

struct SecondIterator{
  MakeIntAndStringIterator int_and_string_iter;
  std::string& operator*(){
    return (*int_and_string_iter).second;
  }
}
```

TRANSFORM ITERATORS INLINED

```
struct SecondIteratorInlined{
  IotaIterator iota_iter;
  std::string& operator*(){
    int i = *iota_iterator;
    return std::pair<int,string>{i*i*i,std::to_string(i)}.second;
  }
}
```

TRANSFORM ITERATORS INLINED

```
struct SecondIteratorInlined{
  IotaIterator iota_iter;
  std::string& operator*(){
    int i = *iota_iterator;
    return std::pair<int,string>{i*i*i,std::to_string(i)}.second;
  }
}
```

TPOIASI

```
int times2(int n){
   return n * 2;
}
bool isMultipleOf4(int n){return n % 4 == 0;}

int main() {
 std::vector<int> numbers = {1, 2, 3, 4, 5};
 auto results = numbers
  | std::ranges::views::transform(times2)
  | std::ranges::views::filter(isMultipleOf4);
 for (auto result : results)
     std::cout << result << '\n';
}
```

- https://www.fluentcpp.com/2019/02/12/the-terrible-problem-of-incrementing-a-smart-iterator/

OUTPUT

```
4
8
```

TPOIASI

```
int times2(int n){
   std::cout << "transforming " << n << "\n";
   return n * 2;
}
bool isMultipleOf4(int n){return n % 4 == 0;}

int main() {
 std::vector<int> numbers = {1, 2, 3, 4, 5};
 auto results = numbers
  | std::ranges::views::transform(times2)
  | std::ranges::views::filter(isMultipleOf4);
 for (auto result : results)
     std::cout << result << '\n';
}
```

- https://www.fluentcpp.com/2019/02/12/the-terrible-problem-of-incrementing-a-smart-iterator/

OUTPUT

```
transforming 1
transforming 2
transforming 2
4
transforming 3
transforming 4
transforming 4
8
transforming 5
```

OUTPUT

```
transforming 1
transforming 2
transforming 2
4
transforming 3
transforming 4
transforming 4
8
transforming 5
```

OUTPUT

```
transforming 1
transforming 2
transforming 2
4
transforming 3
transforming 4
transforming 4
8
transforming 5
```

OUTPUT

```
transforming 1
transforming 2
transforming 2
4
transforming 3
transforming 4
transforming 4
8
transforming 5
```

- The filter iterator has an embedded loop.

STACK SIZE

```
__attribute__((noinline)) void Main7() {
  auto even = [](auto i) { return i % 2 == 0; };
  std::vector<int> r0 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  auto result = r0 | std::views::filter(even) //
                   | std::views::filter(even)    //
                   | std::views::filter(even)    //
                   | std::views::filter(even)    //
                   | std::views::filter(even)    //
                   | std::views::filter(even);
  PrintStack("Main7");
}
```

- https://godbolt.org/z/6zhGaf588

STACK SIZE

```
int main() {
  Main0();
  Main1();
  Main2();
  Main3();
  Main4();
  Main5();
  Main6();
  Main7();
}
```

OUTPUT

```
Main0 192 144 32 0
Main1 192 192 32 0
Main2 192 240 32 0
Main3 192 320 32 0
Main4 192 448 32 0
Main5 192 608 32 0
Main6 192 832 32 0
Main7 192 1120 32 0
```

OUTPUT

```
Main0 192 144 32 0
Main1 192 192 32 0
Main2 192 240 32 0
Main3 192 320 32 0
Main4 192 448 32 0
Main5 192 608 32 0
Main6 192 832 32 0
Main7 192 1120 32 0
```

OUTPUT

```
Main0 192 144 32 0
Main1 192 192 32 0
Main2 192 240 32 0
Main3 192 320 32 0
Main4 192 448 32 0
Main5 192 608 32 0
Main6 192 832 32 0
Main7 192 1120 32 0
```

OUTPUT

```
Main0 192 144 32 0
Main1 192 192 32 0
Main2 192 240 32 0
Main3 192 320 32 0
Main4 192 448 32 0
Main5 192 608 32 0
Main6 192 832 32 0
Main7 192 1120 32 0
```

OUTPUT

```
Main0 192 144 32 0
Main1 192 192 32 0
Main2 192 240 32 0
Main3 192 320 32 0
Main4 192 448 32 0
Main5 192 608 32 0
Main6 192 832 32 0
Main7 192 1120 32 0
```

OUTPUT

```
Main0 192 144 32 0
Main1 192 192 32 0
Main2 192 240 32 0
Main3 192 320 32 0
Main4 192 448 32 0
Main5 192 608 32 0
Main6 192 832 32 0
Main7 192 1120 32 0
```

OUTPUT

```
Main0 192 144 32 0
Main1 192 192 32 0
Main2 192 240 32 0
Main3 192 320 32 0
Main4 192 448 32 0
Main5 192 608 32 0
Main6 192 832 32 0
Main7 192 1120 32 0
```

OUTPUT

```
Main0 192 144 32 0
Main1 192 192 32 0
Main2 192 240 32 0
Main3 192 320 32 0
Main4 192 448 32 0
Main5 192 608 32 0
Main6 192 832 32 0
Main7 192 1120 32 0
```

OUTPUT

```
Main0 192 144 32 0
Main1 192 192 32 0
Main2 192 240 32 0
Main3 192 320 32 0
Main4 192 448 32 0
Main5 192 608 32 0
Main6 192 832 32 0
Main7 192 1120 32 0
```

CUBIC GROWTH

- 144 + ((N+1)(N+2)(N+3)/6 + 2N) * 8 bytes
- h/t Richard Smith

RETHINKING

`<iterator>`

- Lazy
- Pull
- Interactive
- `return`

<algorithm>

```
template< class InputIt, class OutputIt, class UnaryOp >
OutputIt transform( InputIt first1, InputIt last1,
                    OutputIt d_first, UnaryOp unary_op );
```

- Eager
- Push
- Batch
- Continuation passing style - `d_first`

IMPORTANT ANNOUNCEMENT

IMPORTANT ANNOUNCEMENT

Do you or your friends mainly use ranges to:

IMPORTANT ANNOUNCEMENT

Do you or your friends mainly use ranges to:

- Iterate with range for

IMPORTANT ANNOUNCEMENT

Do you or your friends mainly use ranges to:

- Iterate with range for
- Pass to an algorithm

IMPORTANT ANNOUNCEMENT

Do you or your friends mainly use ranges to:

- Iterate with range for
- Pass to an algorithm
- Construct a collection

IMPORTANT ANNOUNCEMENT

Do you or your friends mainly use ranges to:

- Iterate with range for
- Pass to an algorithm
- Construct a collection

YOU MAY BE PAYING TOO MUCH FOR FLEXIBILITY

PREMISE

Trade flexibility of `<iterator>` for safety and efficiency of `<algorithm>`

RETURN VS CONTINUATION PASSING STYLE

```
void Output(const std::string& s){
  std::cout << s << "\n";
}

std::string& Iterator(){
  return std::pair{1, std::string("Hello")}.second;
}

void Unsafe(){
  Output(Iterator());
}
```

RETURN VS CONTINUATION PASSING STYLE

```
void Output(const std::string& s){
  std::cout << s << "\n";
}

std::string& Iterator(){
  return std::pair{1, std::string("Hello")}.second;
}

void Unsafe(){
  Output(Iterator());
}
```

RETURN VS CONTINUATION PASSING STYLE

```
void Output(const std::string& s){
  std::cout << s << "\n";
}

template<typename F>
void Algorithm(F f){
  f(std::pair<int, std::string>(1, "Hello").second);
}

void Safe(){
  Algorithm(&Output);
}
```

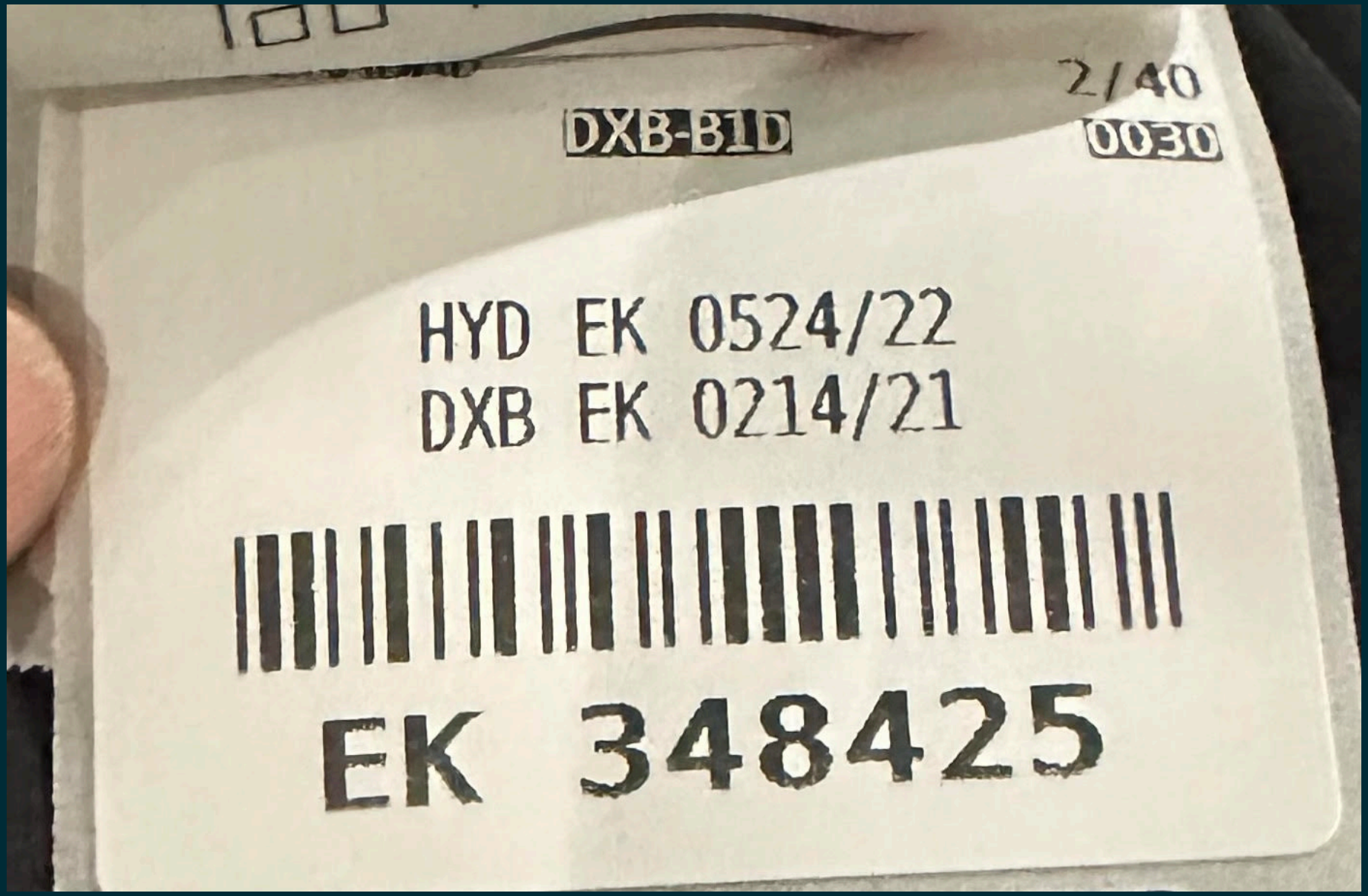
RETURN VS CONTINUATION PASSING STYLE

```
void Output(const std::string& s){
  std::cout << s << "\n";
}

template<typename F>
void Algorithm(F f){
  f(std::pair<int, std::string>(1, "Hello").second);
}

void Safe(){
  Algorithm(&Output);
}
```

LOST LUGGAGE IS A DANGLING REFERENCE


2/40
DXB-BID
0030
HYD EK 0524/22
DXB EK 0214/21
EK 348425

LANDINGS (LIKE RETURNS) ARE AN OPPORTUNITY TO DANGLE REFERENCES


SINGAPORE AIRLINES

RAPPEL

- Google's alterative to std::ranges for algorithm composition
- Based on the eager model of `<algorithm>`
- Passes each value through the series of transformations using continuation passing style.

QUICK EXAMPLES

We will have an example that we look at more in depth coming up. This is mainly just to get a feel of the shape.

CRASH EXAMPLE

```
using IntAndString = std::pair<int, std::string>;
auto make_int_and_string = [](int i) -> IntAndString {
    return {i*i*i, std::to_string(i)};
};
rpl::Apply(rpl::Iota(1, 10000001),
  rpl::Transform(make_int_and_string),
  rpl::Filter([](const auto& p) {
       return p.first >= std::hash<std::string>()(p.second);
   }),
   rpl::Transform(&IntAndString::second),
   rpl::Take(4),
   rpl::ForEach([](const auto& s){
       std::cout << s << "\n";
   })
);
```

OUTPUT

```
1291
1292
1293
1294
```

TPOIASI

```
int times2(int n){
   std::cout << "transforming: " << n << "\n";
   return n * 2;
}
bool isMultipleOf4(int n){return n % 4 == 0;}

int main() {
 std::vector<int> numbers = {1, 2, 3, 4, 5};
 rpl::Apply(numbers
  rpl::Transform(times2),
  rpl::Filter(isMultipleOf4),
  rpl::ForEach([](auto result){
     std::cout << result << '\n';
  }));
}
```

OUTPUT

```
transforming: 1
transforming: 2
4
transforming: 3
transforming: 4
8
transforming: 5
```

SIMPLER EVALUATION MODEL

- Transforms and Filters are not repeatedly evaluated
- Stateful function objects for Transform
- Stateful predicates for Filter

STACK SIZE

```
__attribute__((noinline)) void RappelMain7() {
  auto even = [](auto i) { return i % 2 == 0; };

  std::vector<int> r0 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  [[maybe_unused]] auto result = rpl::Apply(r0,
                                 rpl::Filter(even),
                                 rpl::Filter(even),
                                 rpl::Filter(even),
                                 rpl::Filter(even),
                                 rpl::Filter(even),
                                 rpl::Filter(even),
                                 rpl::Filter(even),
                                 rpl::To<std::vector>());
  PrintStack("RappelMain7");
}
```

OUTPUT

```
RappelMain0 224 112 32 192 0
RappelMain1 224 112 32 192 0
RappelMain2 224 128 32 192 0
RappelMain3 224 128 32 192 0
RappelMain4 224 128 32 192 0
RappelMain5 224 144 32 192 0
RappelMain6 224 160 32 192 0
RappelMain7 224 176 32 192 0
RappelMain14 224 320 32 192 0
```

OUTPUT

```
RappelMain0 224 112 32 192 0
RappelMain1 224 112 32 192 0
RappelMain2 224 128 32 192 0
RappelMain3 224 128 32 192 0
RappelMain4 224 128 32 192 0
RappelMain5 224 144 32 192 0
RappelMain6 224 160 32 192 0
RappelMain7 224 176 32 192 0
RappelMain14 224 320 32 192 0
```

OUTPUT

```
RappelMain0 224 112 32 192 0
RappelMain1 224 112 32 192 0
RappelMain2 224 128 32 192 0
RappelMain3 224 128 32 192 0
RappelMain4 224 128 32 192 0
RappelMain5 224 144 32 192 0
RappelMain6 224 160 32 192 0
RappelMain7 224 176 32 192 0
RappelMain14 224 320 32 192 0
```

OUTPUT

```
RappelMain0 224 112 32 192 0
RappelMain1 224 112 32 192 0
RappelMain2 224 128 32 192 0
RappelMain3 224 128 32 192 0
RappelMain4 224 128 32 192 0
RappelMain5 224 144 32 192 0
RappelMain6 224 160 32 192 0
RappelMain7 224 176 32 192 0
RappelMain14 224 320 32 192 0
```

OUTPUT

```
RappelMain0 224 112 32 192 0
RappelMain1 224 112 32 192 0
RappelMain2 224 128 32 192 0
RappelMain3 224 128 32 192 0
RappelMain4 224 128 32 192 0
RappelMain5 224 144 32 192 0
RappelMain6 224 160 32 192 0
RappelMain7 224 176 32 192 0
RappelMain14 224 320 32 192 0
```

OUTPUT

```
RappelMain0 224 112 32 192 0
RappelMain1 224 112 32 192 0
RappelMain2 224 128 32 192 0
RappelMain3 224 128 32 192 0
RappelMain4 224 128 32 192 0
RappelMain5 224 144 32 192 0
RappelMain6 224 160 32 192 0
RappelMain7 224 176 32 192 0
RappelMain14 224 320 32 192 0
```

OUTPUT

```
RappelMain0 224 112 32 192 0
RappelMain1 224 112 32 192 0
RappelMain2 224 128 32 192 0
RappelMain3 224 128 32 192 0
RappelMain4 224 128 32 192 0
RappelMain5 224 144 32 192 0
RappelMain6 224 160 32 192 0
RappelMain7 224 176 32 192 0
RappelMain14 224 320 32 192 0
```

OUTPUT

```
RappelMain0 224 112 32 192 0
RappelMain1 224 112 32 192 0
RappelMain2 224 128 32 192 0
RappelMain3 224 128 32 192 0
RappelMain4 224 128 32 192 0
RappelMain5 224 144 32 192 0
RappelMain6 224 160 32 192 0
RappelMain7 224 176 32 192 0
RappelMain14 224 320 32 192 0
```

OUTPUT

```
RappelMain0 224 112 32 192 0
RappelMain1 224 112 32 192 0
RappelMain2 224 128 32 192 0
RappelMain3 224 128 32 192 0
RappelMain4 224 128 32 192 0
RappelMain5 224 144 32 192 0
RappelMain6 224 160 32 192 0
RappelMain7 224 176 32 192 0
RappelMain14 224 320 32 192 0
```

OUTPUT

```
RappelMain0 224 112 32 192 0
RappelMain1 224 112 32 192 0
RappelMain2 224 128 32 192 0
RappelMain3 224 128 32 192 0
RappelMain4 224 128 32 192 0
RappelMain5 224 144 32 192 0
RappelMain6 224 160 32 192 0
RappelMain7 224 176 32 192 0
RappelMain14 224 320 32 192 0
```

BENCHMARKS

HANDWRITTEN LOOP

```
std::vector<int> result;
for (int i : v) {
  if (i % 2 != 0 && i % 3 != 0 && i % 5 != 0
      && i % 7 != 0 && i % 11 != 0) {
    result.push_back(i);
  }
}
```

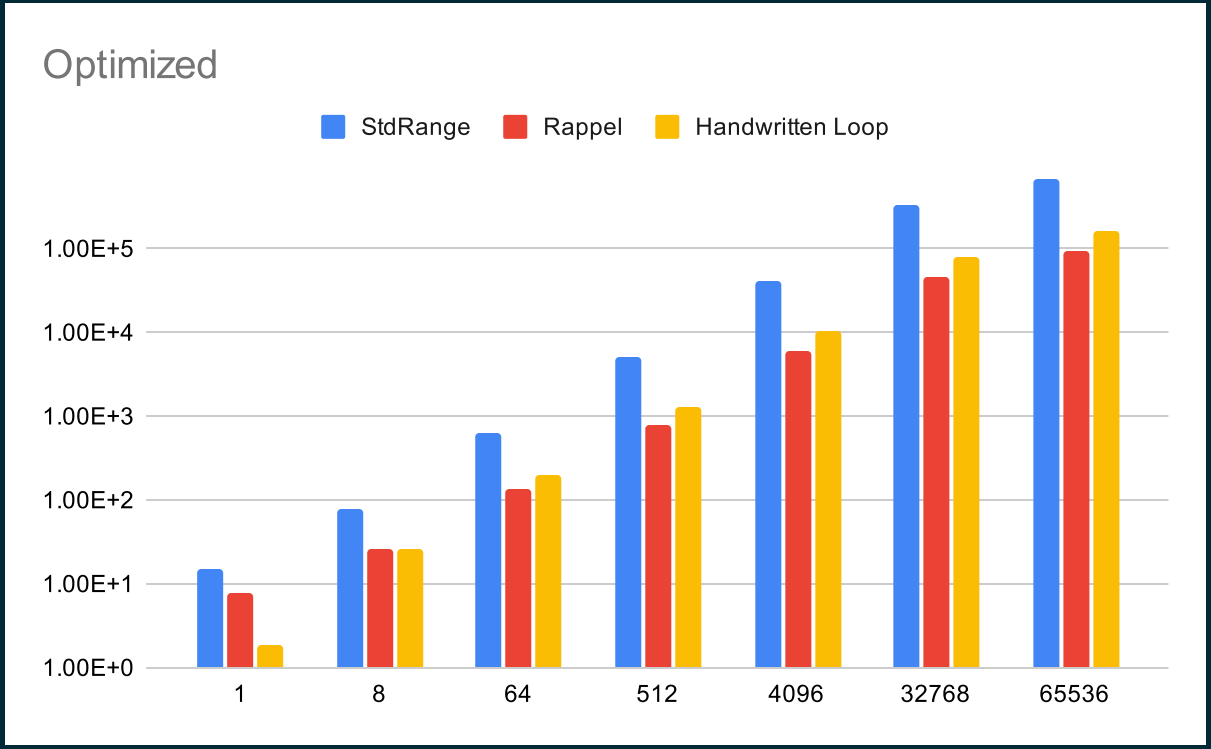
STD::RANGES

```
auto to_vector = [](auto&& r){
  return std::vector<int>(r.begin(), r.end());
};
using std::ranges::views::filter
std::vector<int> result = to_vector(v
 | filter(not_divisible_by_2)
 | filter(not_divisible_by_3)
 | filter(not_divisible_by_5)
 | filter(not_divisible_by_7)
 | filter(not_divisible_by_11));
```

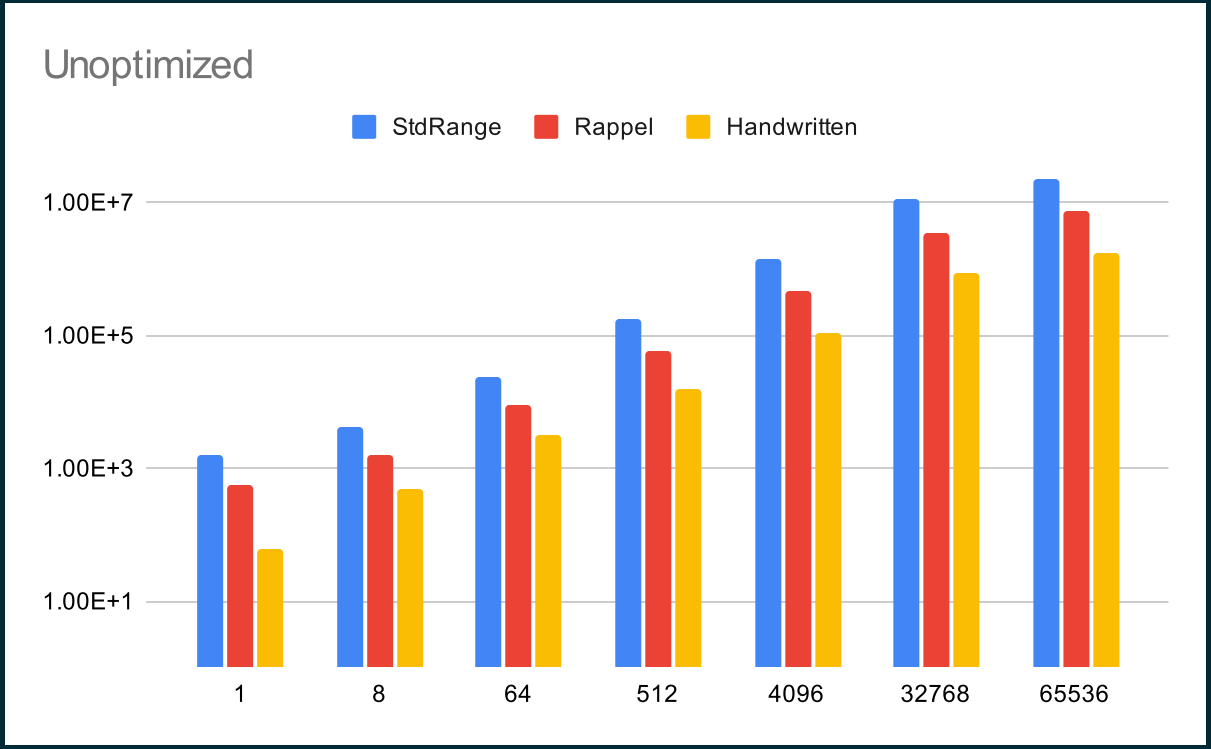
RAPPEL

```
std::vector<int> result = rpl::Apply(v,
     rpl::Filter(not_divisible_by_2),
     rpl::Filter(not_divisible_by_3),
     rpl::Filter(not_divisible_by_5),
     rpl::Filter(not_divisible_by_7),
     rpl::Filter(not_divisible_by_11),
     rpl::To<std::vector>()
 );
```

OPTIMIZED

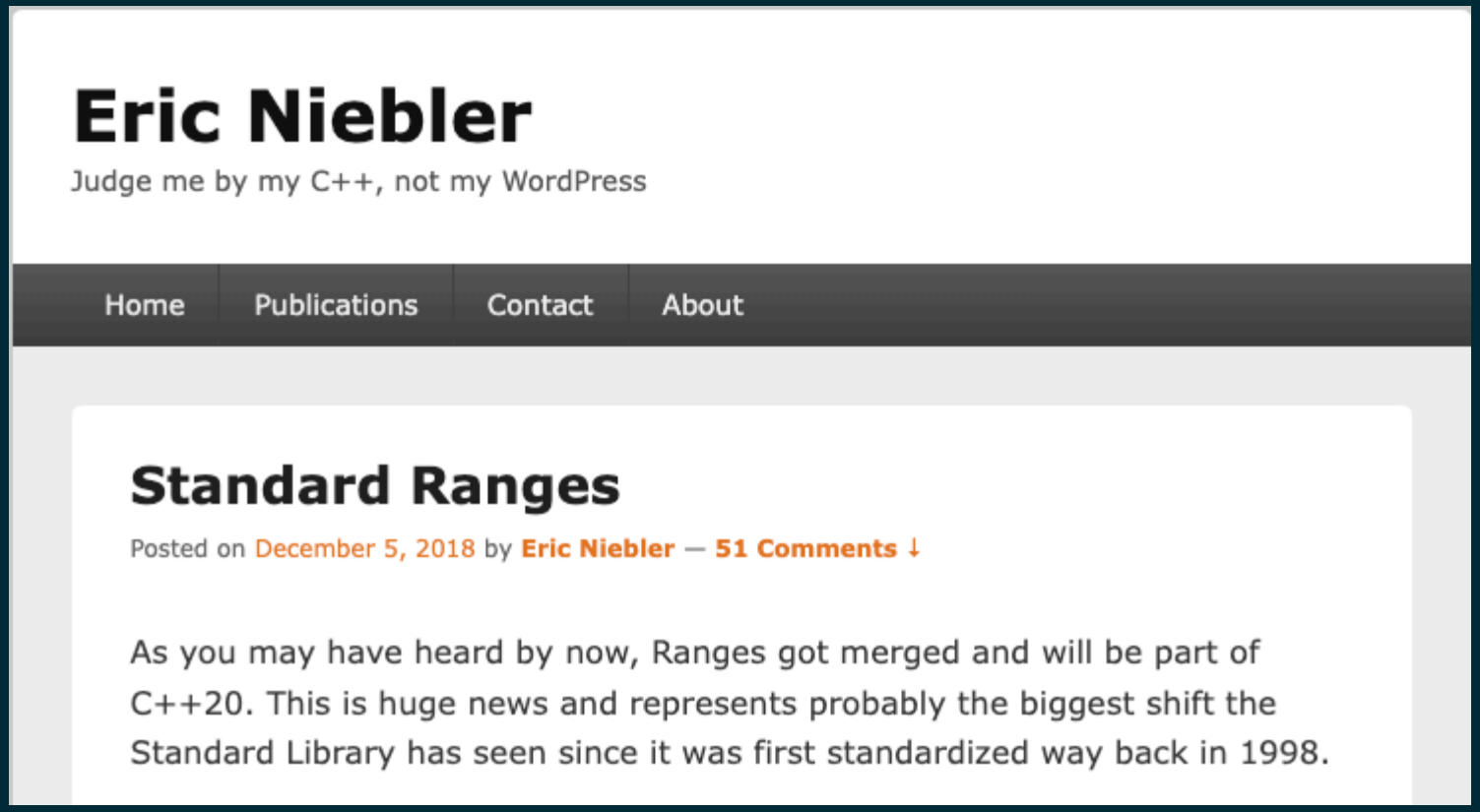

Optimized
StdRange
Rappel
Handwritten Loop
1.00E+5
1.00E+4
1.00E+3
1.00E+2
1.00E+1
1.00E+0
1
8
64
512
4096
32768
65536

UNOPTIMIZED



INTRODUCING RAPPEL

IMITATION IS THE SINCEREST FORM OF FLATTERY



Pythagorian Triples, Revisited

Some years ago now, I wrote a blog post about how to use ranges to generate an infinite list of Pythagorean triples: 3-tuples of integers where the sum of the squares of the first two equals the square of the third.

Below is the complete solution as it will look in standard C++20. I take the solution apart after the break.

STD::RANGES IMPLEMENTATION

HELPER RANGE

```
template<Semiregular T>
struct maybe_view : view_interface<maybe_view<T>> {
  maybe_view() = default;
  maybe_view(T t) : data_(std::move(t)) {
  }
  T const *begin() const noexcept {
    return data_ ? &*data_ : nullptr;
  }
  T const *end() const noexcept {
    return data_ ? &*data_ + 1 : nullptr;
  }
private:
  optional<T> data_{};
};
```

HELPER LAMBDAS

```
inline constexpr auto for_each =
  []<Range R,
     Iterator I = iterator_t<R>,
     IndirectUnaryInvocable<I> Fun>(R&& r, Fun fun)
        requires Range<indirect_result_t<Fun, I>> {
      return std::forward<R>(r)
        | view::transform(std::move(fun))
        | view::join;
  };

inline constexpr auto yield_if =
  []<Semiregular T>(bool b, T x) {
    return b ? maybe_view{std::move(x)}
             : maybe_view<T>{};
  };
```

LAZY TRIPLES

```
using view::iota;
auto triples =
  for_each(iota(1), [](int z) {
    return for_each(iota(1, z+1), [=](int x) {
      return for_each(iota(x, z+1), [=](int y) {
        return yield_if(x*x + y*y == z*z,
          make_tuple(x, y, z));
      });
    });
  });
```

OUTPUT

```
for(auto triple : triples | view::take(10)) {
  cout << '('
       << get<0>(triple) << ','
       << get<1>(triple) << ','
       << get<2>(triple) << ')' << '\n';
}
```

RAPPEL

PYTHAGOREAN TRIPLES

```
void OutputPythagoreanTriples() {
 rpl::Apply(
  rpl::Iota(1),
  rpl::ZipResult([](int c) {return rpl::Iota(1, c+1);}),
  rpl::Flatten(),
  rpl::ZipResult([](int c, int a){return rpl::Iota(a, c+1);}),
  rpl::Flatten(),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
   std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

PYTHAGOREAN TRIPLES

```
void OutputPythagoreanTriples() {
 rpl::Apply(
  rpl::Iota(1),
  rpl::ZipResult([](int c) {return rpl::Iota(1, c+1);}),
  rpl::Flatten(),
  rpl::ZipResult([](int c, int a){return rpl::Iota(a, c+1);}),
  rpl::Flatten(),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
   std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

PYTHAGOREAN TRIPLES

```
void OutputPythagoreanTriples() {
 rpl::Apply(
  rpl::Iota(1),
  rpl::ZipResult([](int c) {return rpl::Iota(1, c+1);}),
  rpl::Flatten(),
  rpl::ZipResult([](int c, int a){return rpl::Iota(a, c+1);}),
  rpl::Flatten(),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
   std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

PYTHAGOREAN TRIPLES

```
void OutputPythagoreanTriples() {
 rpl::Apply(
  rpl::Iota(1),
  rpl::ZipResult([](int c) {return rpl::Iota(1, c+1);}),
  rpl::Flatten(),
  rpl::ZipResult([](int c, int a){return rpl::Iota(a, c+1);}),
  rpl::Flatten(),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
   std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

PYTHAGOREAN TRIPLES

```
void OutputPythagoreanTriples() {
 rpl::Apply(
  rpl::Iota(1),
  rpl::ZipResult([](int c) {return rpl::Iota(1, c+1);}),
  rpl::Flatten(),
  rpl::ZipResult([](int c, int a){return rpl::Iota(a, c+1);}),
  rpl::Flatten(),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
   std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

PYTHAGOREAN TRIPLES

```
void OutputPythagoreanTriples() {
 rpl::Apply(
  rpl::Iota(1),
  rpl::ZipResult([](int c) {return rpl::Iota(1, c+1);}),
  rpl::Flatten(),
  rpl::ZipResult([](int c, int a){return rpl::Iota(a, c+1);}),
  rpl::Flatten(),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
   std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

PYTHAGOREAN TRIPLES

```
void OutputPythagoreanTriples() {
 rpl::Apply(
  rpl::Iota(1),
  rpl::ZipResult([](int c) {return rpl::Iota(1, c+1);}),
  rpl::Flatten(),
  rpl::ZipResult([](int c, int a){return rpl::Iota(a, c+1);}),
  rpl::Flatten(),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
   std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

PYTHAGOREAN TRIPLES

```
void OutputPythagoreanTriples() {
 rpl::Apply(
  rpl::Iota(1),
  rpl::ZipResult([](int c) {return rpl::Iota(1, c+1);}),
  rpl::Flatten(),
  rpl::ZipResult([](int c, int a){return rpl::Iota(a, c+1);}),
  rpl::Flatten(),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
   std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

PYTHAGOREAN TRIPLES

```
void OutputPythagoreanTriples() {
 rpl::Apply(
  rpl::Iota(1),
  rpl::ZipResult([](int c) {return rpl::Iota(1, c+1);}),
  rpl::Flatten(),
  rpl::ZipResult([](int c, int a){return rpl::Iota(a, c+1);}),
  rpl::Flatten(),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
   std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

PYTHAGOREAN TRIPLES

```
void OutputPythagoreanTriples() {
 rpl::Apply(
  rpl::Iota(1),
  rpl::ZipResult([](int c) {return rpl::Iota(1, c+1);}),
  rpl::Flatten(),
  rpl::ZipResult([](int c, int a){return rpl::Iota(a, c+1);}),
  rpl::Flatten(),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
   std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

PYTHAGOREAN TRIPLES

```
void OutputPythagoreanTriples() {
 rpl::Apply(
  rpl::Iota(1),
  rpl::ZipResult([](int c) {return rpl::Iota(1, c+1);}),
  rpl::Flatten(),
  rpl::ZipResult([](int c, int a){return rpl::Iota(a, c+1);}),
  rpl::Flatten(),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
   std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

REPETITION

```
void OutputPythagoreanTriples() {
 rpl::Apply(
  rpl::Iota(1),
  rpl::ZipResult([](int c) {return rpl::Iota(1, c+1);}),
  rpl::Flatten(),
  rpl::ZipResult([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Flatten(),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
   std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

REPETITION

```
void OutputPythagoreanTriples() {
 rpl::Apply(
  rpl::Iota(1),
  rpl::ZipResult([](int c) {return rpl::Iota(1, c+1);}),
  rpl::Flatten(),
  rpl::ZipResult([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Flatten(),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
   std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

REPETITION

```
void OutputPythagoreanTriples() {
 rpl::Apply(
  rpl::Iota(1),
  rpl::ZipResult([](int c) {return rpl::Iota(1, c+1);}),
  rpl::Flatten(),
  rpl::ZipResult([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Flatten(),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
   std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

COMPOSE

```
void OutputPythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};
 rpl::Apply(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
   std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

COMPOSE

```
void OutputPythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};
 rpl::Apply(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
   std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

COMPOSE

```
void OutputPythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};
 rpl::Apply(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
   std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

FLEXIBILITY

```
void OutputPythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};
 rpl::Apply(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
      std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

FLEXIBILITY

```
void OutputPythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};
 rpl::Apply(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
      std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

FLEXIBILITY

```
void OutputPythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};
 rpl::Apply(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
      std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

FLEXIBILITY

```
void OutputPythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};
 rpl::Apply(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
      std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

FLEXIBILITY

```
auto PythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};
 rpl::Apply(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
      std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

FLEXIBILITY

```
auto PythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};
 rpl::Apply(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
      std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

FLEXIBILITY

```
auto PythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};
 rpl::Apply(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
      std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

FLEXIBILITY

```
auto PythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};
 return rpl::Compose(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
      std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

FLEXIBILITY

```
auto PythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};
 return rpl::Compose(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
      std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

FLEXIBILITY

```
auto PythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};
 return rpl::Compose(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>(),
  rpl::Take(10),
  rpl::ForEach([](int a, int b, int c){
      std::cout << a << " " << b << " " << c << "\n";
  })
 );
}
```

FLEXIBILITY

```
auto PythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};
 return rpl::Compose(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>()




 );
}
```

PYTHAGOREAN TRIPLES

```
auto PythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};

 return rpl::Compose(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>()
 );
}
```

PYTHAGOREAN TRIPLES

```
auto PythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};

 return rpl::Compose(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>()
 );
}
```

PYTHAGOREAN TRIPLES

```
auto PythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};

 return rpl::Compose(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>()
 );
}
```

PYTHAGOREAN TRIPLES

```
auto PythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};

 return rpl::Compose(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>()
 );
}
```

PYTHAGOREAN TRIPLES

```
auto PythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};

 return rpl::Compose(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>()
 );
}
```

PYTHAGOREAN TRIPLES

```
auto PythagoreanTriples() {
 auto zip_flat = [](auto f){
  return rpl::Compose(rpl::ZipResult(f), rpl::Flatten());};

 return rpl::Compose(
  rpl::Iota(1),
  zip_flat([](int c) {return rpl::Iota(1, c+1);}),
  zip_flat([](int c, int a) {return rpl::Iota(a, c+1);}),
  rpl::Filter([](int c, int a, int b){return a*a + b*b == c*c;}),
  rpl::Swizzle<1, 2, 0>()
 );
}
```

PYTHAGOREAN TRIPLES

```
vector<tuple<int,int,int>> triples =
  rpl::Apply(
    PythagoreanTriples(),
    rpl::Take(100),
    rpl::MakeTuple(),
    rpl::To<std::vector>()
  );
```

PYTHAGOREAN TRIPLES

```
vector<tuple<int,int,int>> triples =
  rpl::Apply(
    PythagoreanTriples(),
    rpl::Take(100),
    rpl::MakeTuple(),
    rpl::To<std::vector>()
  );
```

PYTHAGOREAN TRIPLES

```
vector<tuple<int,int,int>> triples =
  rpl::Apply(
    PythagoreanTriples(),
    rpl::Take(100),
    rpl::MakeTuple(),
    rpl::To<std::vector>()
  );
```

PYTHAGOREAN TRIPLES

```
vector<tuple<int,int,int>> triples =
  rpl::Apply(
    PythagoreanTriples(),
    rpl::Take(100),
    rpl::MakeTuple(),
    rpl::To<std::vector>()
  );
```

PYTHAGOREAN TRIPLES

```
vector<tuple<int,int,int>> triples =
  rpl::Apply(
    PythagoreanTriples(),
    rpl::Take(100),
    rpl::MakeTuple(),
    rpl::To<std::vector>()
  );
```

PYTHAGOREAN TRIPLES

```
vector<tuple<int,int,int>> triples =
  rpl::Apply(
    PythagoreanTriples(),
    rpl::Take(100),
    rpl::MakeTuple(),
    rpl::To<std::vector>()
  );
```

RAPPEL BACKGROUND

TERMINOLOGY

- Pipeline
- Stage
- Processing style

PROCESSING STYLE

- How a stage processes elements
- Two kinds
  - Incremental
  - Complete

INCREMENTAL

- Processes elements one at a time
- `Filter`, `Transform` are all examples of *incremental*
- We can evaluate an element and output the result immediately without waiting for other elements

COMPLETE

- Process a single element as a whole
- Sorting is an example of *complete*
- You need all the elements to sort
- Surprising
  - `Max`
  - You take all the elements and reduce them to a single value.
  - You are treating that value as a whole, not part of something else.

INPUT AND OUTPUT

- The input and output processing style don't have to be the same.
- Four possiblities

Input	Output
Incremental	Incremental
Incremental	Complete
Complete	Complete
Complete	Incremental

INCREMENTAL -> INCREMENTAL

- `Transform`
- `Filter`
- Take a single element, and output an element as part of something larger

INCREMENTAL -> COMPLETE

- `To<std::vector>`
- `Max`
- `AnyOf`, `AllOf`, `NoneOf`
- Take values that are parts of a whole and collect them into new whole
- `Accumulate` is the generalization of transformation

COMPLETE -> COMPLETE

- `Sort` (based on `std::sort`)
- Transforms a collection, sorts it, and outputs that collection as a whole
- Others include `PartialSort`, `StableSort`, `NthElement`, `Unique`

COMPLETE -> INCREMENTAL

- Iterating a range
- Implicit in Rappel

RULES FOR INCREMENTAL AND COMPLETE

- The input processing style of a stage has to match the output processing style of the previous stage
  - We can implicitly convert *complete* to *incremental*
- The final stage passed to `Apply` has to be *complete*
  - `Apply` needs to return a single value that stands on its own.

APPLY

```
template <typename Range, typename... Stages>
[[nodiscard]] auto Apply(Range&& range, Stages&&... stages);
```

- The first parameter is a `range`
- The `stages` is one or more of the above stages.
- Note that the return type is `auto`. It will return a value and not a reference. This is for safety by default.

FUME HOODS (LIKE APPLY) ISOLATE DANGEROUS INTERMEDIATES (LIKE REFERENCES TO TEMPORARIES)

TEST

- `std::vector<int> v{...};`
- Assume we capture the return value

```
Apply(v, Max());
Apply(v,Transform(triple));
Apply(v,Transform(triple),To<std::vector>());
Apply(v,Transform(triple),Sort());
Apply(v,Transform(triple),To<std::vector>(),Sort());
Apply(v,Transform(triple),To<std::vector>(),
    Sort(), FilterDuplicates(), Accumulate());
```

TEST

- `std::vector<int> v{...};`
- Assume we capture the return value

```
Apply(v, Max());
Apply(v,Transform(triple));
Apply(v,Transform(triple),To<std::vector>());
Apply(v,Transform(triple),Sort());
Apply(v,Transform(triple),To<std::vector>(),Sort());
Apply(v,Transform(triple),To<std::vector>(),
    Sort(), FilterDuplicates(), Accumulate());
```

TEST

- `std::vector<int> v{...};`
- Assume we capture the return value

```
Apply(v, Max());
Apply(v,Transform(triple));
Apply(v,Transform(triple),To<std::vector>());
Apply(v,Transform(triple),Sort());
Apply(v,Transform(triple),To<std::vector>(),Sort());
Apply(v,Transform(triple),To<std::vector>(),
    Sort(), FilterDuplicates(), Accumulate());
```

TEST

- `std::vector<int> v{...};`
- Assume we capture the return value

```
Apply(v, Max());
Apply(v,Transform(triple));
Apply(v,Transform(triple),To<std::vector>());
Apply(v,Transform(triple),Sort());
Apply(v,Transform(triple),To<std::vector>(),Sort());
Apply(v,Transform(triple),To<std::vector>(),
    Sort(), FilterDuplicates(), Accumulate());
```

TEST

- `std::vector<int> v{...};`
- Assume we capture the return value

```
Apply(v, Max());
Apply(v,Transform(triple));
Apply(v,Transform(triple),To<std::vector>());
Apply(v,Transform(triple),Sort());
Apply(v,Transform(triple),To<std::vector>(),Sort());
Apply(v,Transform(triple),To<std::vector>(),
    Sort(), FilterDuplicates(), Accumulate());
```

TEST

- `std::vector<int> v{...};`
- Assume we capture the return value

```
Apply(v, Max());
Apply(v,Transform(triple));
Apply(v,Transform(triple),To<std::vector>());
Apply(v,Transform(triple),Sort());
Apply(v,Transform(triple),To<std::vector>(),Sort());
Apply(v,Transform(triple),To<std::vector>(),
    Sort(), FilterDuplicates(), Accumulate());
```

COMPOSE

- Creates new stages by composition
- It is just a thin wrapper over `std::tuple`.
- `Apply` treats passing in `Compose` as if you passed what is in the elements directly.

STAGES

AIM FOR DEFAULT SAFETY

- `Max` return `optional<T>`
- `PartialSort(n)` is safe even if `n` > number of elements
- `NthElement(n)` is safe even if `n` > number of elements

COMPLETE TRANSFORMS

- TransformComplete
- Sort
- StableSort
- PartialSort
- NthElement
- Unique

INCREMENTAL TO COMPLETE ACCUMULATORS

- Accumulate
- AccumulateInPlace
- ForEach
- To
- ToMap
- Into

ACCUMULATORS (CONTINUED)

- Front
- Count
- Min
- Max
- MinMax
- AnyOf
- AllOf
- NoneOf

FILTERS

- Filter
- Take
- TakeWhile
- Skip
- SkipWhile
- FilterDuplicates

INCREMENTAL TRANSFORMS

- Transform
- TransformArg
- Ref
- Move
- LRef
- Deref
- AddressOf
- Get
- Cast
- Construct

INCREMENTAL ARGUMENT MANIPULATION

- ExpandTuple
- Enumerate
- ZipWith
- ZipResult
- CartesianProductWith
- Swizzle
- MakeTuple
- MakePair
- Flatten

FLAT MAP

- FlatMap
- Can output zero to many results

HIGHER ORDER

- GroupBy
- MapGroupBy
- Chunk
- Tee

SEQUENCE EXTENSION

- RepeatWhile
- RepeatN
- ChainBefore
- ChainAfter

ERROR HANDLING

- UnwrapOptional
- UnwrapOptionalArg
- UnwrapOptionalComplete

IMPLEMENTING STAGES

- Warning: Internals ahead
- Simplifications and hand-waving ahead

STAGE IMPL

```
template<typename InputType, typename... OtherParameters>
struct FooStageImpl{
    using OutputType = ;

    // Required for incremental
    template<typename Next>
    void ProcessIncremental(InputType input, Next&& next);
    // Optional for incremental
    template<typename Next>
    decltype(auto) End(Next&& next);
    bool Done() const;

    // Required for complete
    decltype(auto) ProcessComplete(InputType input, Next&& next);
};
```

COMPLETE TRANSFORM

```
template <typename InputType, typename F>
struct TransformCompleteImpl {
  F f;

  using OutputType = std::invoke_result_t<F,InputType>;

  template <typename Next>
  decltype(auto) ProcessComplete(InputType input, Next&& next) {
    return next.ProcessComplete(std::invoke(f,
      static_cast<InputType>(input)));
  }
};
```

INCREMENTAL TRANSFORM

```
template <typename InputType, typename F>
struct TransformImpl {
  using OutputType = std::invoke_result_t<F,InputType>;

  template <typename Next>
  decltype(auto) ProcessIncremental(InputType input, Next&& next)
    return next.ProcessIncremental(std::invoke(f,
      static_cast<InputType>(input)));
  }
};
```

FILTER

```
template <typename InputType, typename Predicate>
struct FilterImplImpl<InputType, Predicate> {
  using OutputType = InputType;
  Predicate f;

  template <typename Next>
  void ProcessIncremental(InputType input, Next&& next) {
    if(std::invoke(f,std::as_const(input))){
        next.ProcessIncremental(static_cast<InputType>(input));
    }
};
```

HOLDING IMPLS

```
template <typename Chain, size_t I, typename InputTypeParam,
          typename Stage>
class StageChainComponent {
 public:
  using InputType = InputTypeParam;
  using OutputType = typename Stage::OutputType;
private:
  Stage stage_;
public:
  decltype(auto) Get(Index<I>) { return *this; }
  auto& GetChain() { return static_cast<Chain&>(*this); }
  decltype(auto) Next() { return GetChain().Get(Index<I + 1>());
```

```
constexpr decltype(auto) End() {
  if constexpr (kHasEnd<Stage, decltype(Next())>) {
    return stage_.End(Next());
  } else {
    return Next().End();
  }
}
constexpr bool Done() const {
  if constexpr (kHasDone<Stage>) {
    return stage_.Done();
  } else {
    return false;
  }
}
```

```
  void ProcessIncremental(InputType t) {
    stage_.ProcessIncremental(static_cast<InputType>(t), Next());
  }
  decltype(auto) ProcessComplete(InputType t) {
    if constexpr(is_incremental){
     for(auto&& v: t) // Handwavy
       stage_.ProcessIncremental(v,Next());
     return End();
    }else{
      return
       stage_.ProcessComplete(static_cast<InputType>(t), Next());
     }
  }
};
```

CHAIN

```
template<size_t... Is,typename... Components>
struct Chain<std::index_sequence<Is...>,Components...>
:ComponentT<Chain,Is,Components>...{
    using ComponentT<Chain,Is,Components>::Get...;

    template<typename T>
    auto ProcessComplete(T&& t){
        return std::forward<T>(t);
    }

    auto& Get(std::Index<sizeof...(Is)>){
        return *this;
    }
};
```

```
template<typename Range, typename... Stages>
auto Apply(Range&& range, Stages&&... stages){
    auto chain = MakeChain(std::forward<Stages>(stages)...);
    return chain.Get(Index<0>())
      .ProcessComplete(std::forward<Range>(range));
}
```

USING RAPPEL

COMPOSITION

`Compose` is a first-class stage

```
auto FlattenOptional() {
  return rpl::Compose(rpl::Filter(), rpl::Deref());
}

rpl::Apply(
  std::vector<std::optional<int>>{1, std::nullopt, 3},
  FlattenOptional(),
  rpl::To<std::vector>());
```

`Compose` is a first-class stage

```
auto FlattenOptional() {
  return rpl::Compose(rpl::Filter(), rpl::Deref());
}

rpl::Apply(
  std::vector<std::optional<int>>{1, std::nullopt, 3},
  FlattenOptional(),
  rpl::To<std::vector>());
```

`Compose` is a first-class stage

```
auto FlattenOptional() {
  return rpl::Compose(rpl::Filter(), rpl::Deref());
}

rpl::Apply(
  std::vector<std::optional<int>>{1, std::nullopt, 3},
  FlattenOptional(),
  rpl::To<std::vector>());
```

Compose generators

```
auto PerfectSquares() {
  return rpl::Compose(
    rpl::Iota(1),
    rpl::Transform([](int i) { return i*i; }));
}

rpl::Apply(
  PerfectSquares(),
  rpl::Filter([](int i) { return i >= 100; }),
  rpl::Take(5),
  rpl::To<std::vector>());
```

Compose generators

```
auto PerfectSquares() {
  return rpl::Compose(
    rpl::Iota(1),
    rpl::Transform([](int i) { return i*i; }));
}

rpl::Apply(
  PerfectSquares(),
  rpl::Filter([](int i) { return i >= 100; }),
  rpl::Take(5),
  rpl::To<std::vector>());
```

Compose generators

```
auto PerfectSquares() {
  return rpl::Compose(
    rpl::Iota(1),
    rpl::Transform([](int i) { return i*i; }));
}

rpl::Apply(
  PerfectSquares(),
  rpl::Filter([](int i) { return i >= 100; }),
  rpl::Take(5),
  rpl::To<std::vector>());
```

Compose ranges

```
struct Transactions {
  std::vector<double> amounts;

  auto Credits() const {
    return rpl::Compose(
      std::cref(amounts),
      rpl::Filter([](double d) { return d > 0; }));
  }
};
```

Compose ranges

```
struct Transactions {
  std::vector<double> amounts;

  auto Credits() const {
    return rpl::Compose(
      std::cref(amounts),
      rpl::Filter([](double d) { return d > 0; }));
  }
};
```

GENERATORS

- Input iterators can be difficult to write
- Want an easy way to make generators such as `Iota`

```
template <typename T, typename GeneratorInvocable>
auto Generator(GeneratorInvocable&& generator_invocable);
```

IOTA

```
template <typename T>
auto Iota(T begin) {
  return rpl::Generator<T>([begin](auto output) mutable {
    std::move(output)(begin);
    ++begin;
  });
};
```

IOTA

```
template <typename T>
auto Iota(T begin) {
  return rpl::Generator<T>([begin](auto output) mutable {
    std::move(output)(begin);
    ++begin;
  });
};
```

IOTA WITH END

```
template <typename T, End>
auto Iota(T begin, End end) {
  return rpl::Generator<T>([begin,end](auto output) mutable {
    if(begin != end){
        std::move(output)(begin);
        ++begin;
    }
  });
};
```

IOTA WITH END

```
template <typename T, End>
auto Iota(T begin, End end) {
  return rpl::Generator<T>([begin,end](auto output) mutable {
    if(begin != end){
        std::move(output)(begin);
        ++begin;
    }
  });
};
```

MULTI-ARGUMENT STREAMS

Multi-argument streams are passed as seperate arguments

```
std::map<int, std::vector<int>> values = {...};
rpl::Apply(values,                  // pair<int, vector>
           rpl::ExpandTuple(),  // int, vector
           rpl::Flatten(),      // int, int
           rpl::ForEach([](int k, int v) { ... }));
```

Multi-argument streams are passed as seperate arguments

```
std::map<int, std::vector<int>> values = {...};
rpl::Apply(values,                  // pair<int, vector>
           rpl::ExpandTuple(),  // int, vector
           rpl::Flatten(),      // int, int
           rpl::ForEach([](int k, int v) { ... }));
```

Multi-argument streams are passed as seperate arguments

```
std::map<int, std::vector<int>> values = {...};
rpl::Apply(values,                  // pair<int, vector>
           rpl::ExpandTuple(),  // int, vector
           rpl::Flatten(),      // int, int
           rpl::ForEach([](int k, int v) { ... }));
```

Multi-argument streams are passed as seperate arguments

```
std::map<int, std::vector<int>> values = {...};
rpl::Apply(values,                  // pair<int, vector>
           rpl::ExpandTuple(),      // int, vector
           rpl::Flatten(),          // int, int
           rpl::ForEach([](int k, int v) { ... }));
```

Multi-argument streams are passed as seperate arguments

```
std::map<int, std::vector<int>> values = {...};
rpl::Apply(values,                  // pair<int, vector>
           rpl::ExpandTuple(),      // int, vector
           rpl::Flatten(),          // int, int
           rpl::ForEach([](int k, int v) { ... }));
```

First class support

```
std::map<std::string, int> name2id = {...};
auto id2name = rpl::Apply(
  std::move(name2id),
  rpl::ExpandTuple(),
  rpl::Swizzle<1, 0>(),
  rpl:ToMap<std::map>());
```

First class support

```
std::map<std::string, int> name2id = {...};
auto id2name = rpl::Apply(
  std::move(name2id),
  rpl::ExpandTuple(),
  rpl::Swizzle<1, 0>(),
  rpl:ToMap<std::map>());
```

TransformArg

```
std::map<std::string, Person> name2person = {...};
auto id2name = rpl::Apply(
  std::move(name2id),
  rpl::ExpandTuple(),
  rpl::TransformArg<1>(&Person::id),
  rpl::Swizzle<1, 0>(),
  rpl:ToMap<std::map>());
```

TransformArg

```
std::map<std::string, Person> name2person = {...};
auto id2name = rpl::Apply(
  std::move(name2id),
  rpl::ExpandTuple(),
  rpl::TransformArg<1>(&Person::id),
  rpl::Swizzle<1, 0>(),
  rpl:ToMap<std::map>());
```

Preserve Reference Semantics

```
std::vector<std::unique_ptr<int>> values = {...};
rpl::Apply(
  values,
  rpl::ZipResult([](const auto& ptr) {
    return std::make_unique<int>(*ptr); }),
  rpl::ForEach([](auto&& first, auto&& second) {
      static_assert(std::is_same_v<
        decltype(first),
        const std::unique_ptr<int>&
      >);
      static_assert(std::is_same_v<
        decltype(second),
        std::unique_ptr<int>&&
      >);
    }));
```

Preserve Reference Semantics

```
std::vector<std::unique_ptr<int>> values = {...};
rpl::Apply(
  values,
  rpl::ZipResult([](const auto& ptr) {
    return std::make_unique<int>(*ptr); }),
  rpl::ForEach([](auto&& first, auto&& second) {
      static_assert(std::is_same_v<
        decltype(first),
        const std::unique_ptr<int>&
      >);
      static_assert(std::is_same_v<
        decltype(second),
        std::unique_ptr<int>&&
      >);
    }));
```

Preserve Reference Semantics

```
std::vector<std::unique_ptr<int>> values = {...};
rpl::Apply(
  values,
  rpl::ZipResult([](const auto& ptr) {
    return std::make_unique<int>(*ptr); }),
  rpl::ForEach([](auto&& first, auto&& second) {
      static_assert(std::is_same_v<
        decltype(first),
        const std::unique_ptr<int>&
      >);
      static_assert(std::is_same_v<
        decltype(second),
        std::unique_ptr<int>&&
      >);
    }));
```

ERROR HANDLING

Short-circuit on error and wrap results

```
std::optional<int> ParseInt(std::string_view);

std::vector<std::optional<std::string>> values = {...};
std::optional<int> sum = rpl::Apply(
  values,                          // optional<string>
  rpl::UnwrapOptional(),           // string
  rpl::Transform(&ParseInt),       // optional<int>
  rpl::UnwrapOptional(),           // int
  rpl::Accumulate()                // int
);                                 // optional<int>
```

Short-circuit on error and wrap results

```
std::optional<int> ParseInt(std::string_view);

std::vector<std::optional<std::string>> values = {...};
std::optional<int> sum = rpl::Apply(
  values,                          // optional<string>
  rpl::UnwrapOptional(),           // string
  rpl::Transform(&ParseInt),       // optional<int>
  rpl::UnwrapOptional(),           // int
  rpl::Accumulate()                // int
);                                 // optional<int>
```

Short-circuit on error and wrap results

```
std::optional<int> ParseInt(std::string_view);

std::vector<std::optional<std::string>> values = {...};
std::optional<int> sum = rpl::Apply(
  values,                          // optional<string>
  rpl::UnwrapOptional(),           // string
  rpl::Transform(&ParseInt),       // optional<int>
  rpl::UnwrapOptional(),           // int
  rpl::Accumulate()                // int
);                                 // optional<int>
```

Short-circuit on error and wrap results

```
std::optional<int> ParseInt(std::string_view);

std::vector<std::optional<std::string>> values = {...};
std::optional<int> sum = rpl::Apply(
  values,                          // optional<string>
  rpl::UnwrapOptional(),           // string
  rpl::Transform(&ParseInt),       // optional<int>
  rpl::UnwrapOptional(),           // int
  rpl::Accumulate()                // int
);                                 // optional<int>
```

Short-circuit on error and wrap results

```
std::optional<int> ParseInt(std::string_view);

std::vector<std::optional<std::string>> values = {...};
std::optional<int> sum = rpl::Apply(
  values,                          // optional<string>
  rpl::UnwrapOptional(),           // string
  rpl::Transform(&ParseInt),       // optional<int>
  rpl::UnwrapOptional(),           // int
  rpl::Accumulate()                // int
);                                 // optional<int>
```

Short-circuit on error and wrap results

```
std::optional<int> ParseInt(std::string_view);

std::vector<std::optional<std::string>> values = {...};
std::optional<int> sum = rpl::Apply(
  values,                          // optional<string>
  rpl::UnwrapOptional(),           // string
  rpl::Transform(&ParseInt),       // optional<int>
  rpl::UnwrapOptional(),           // int
  rpl::Accumulate()                // int
);                                 // optional<int>
```

Monadic style

```
std::optional<int> ParseInt(std::string_view);
int Squared(int i);

std::optional<std::string> value = ...;

std::optional<int> cpp23 = value
  .and_then(&ParseInt)
  .transform(&Squared);

std::optional<int> result = rpl::Apply(
  value,
  rpl::UnwrapOptionalComplete(),
  rpl::TransformComplete(&ParseInt),
  rpl::TransformComplete(&Squared));
```

Monadic style

```
std::optional<int> ParseInt(std::string_view);
int Squared(int i);

std::optional<std::string> value = ...;

std::optional<int> cpp23 = value
  .and_then(&ParseInt)
  .transform(&Squared);

std::optional<int> result = rpl::Apply(
  value,
  rpl::UnwrapOptionalComplete(),
  rpl::TransformComplete(&ParseInt),
  rpl::TransformComplete(&Squared));
```

Monadic style

```
std::optional<int> ParseInt(std::string_view);
int Squared(int i);

std::optional<std::string> value = ...;

std::optional<int> cpp23 = value
  .and_then(&ParseInt)
  .transform(&Squared);

std::optional<int> result = rpl::Apply(
  value,
  rpl::UnwrapOptionalComplete(),
  rpl::TransformComplete(&ParseInt),
  rpl::TransformComplete(&Squared));
```

Monadic StatusOr

```
absl::StatusOr<int> ParseInt(std::string_view);
int Squared(int i);

absl::StatusOr<std::vector<std::string>> values = ...;

absl::StatusOr<std::vector<int>> result = rpl::Apply(
  values,                              // StatusOr<vector<string>>
  rpl::UnwrapStatusOrComplete(), // std::vector
  rpl::Transform(&ParseInt),     // StatusOr<int>
  rpl::UnwrapStatusOr(),         // int
  rpl::Transform(&Squared),      // int
  rpl::To<std::vector>()         // vector<int>
);                                     // StatusOr<vector<int>>
```

Monadic StatusOr

```
absl::StatusOr<int> ParseInt(std::string_view);
int Squared(int i);

absl::StatusOr<std::vector<std::string>> values = ...;

absl::StatusOr<std::vector<int>> result = rpl::Apply(
  values,                              // StatusOr<vector<string>>
  rpl::UnwrapStatusOrComplete(), // std::vector
  rpl::Transform(&ParseInt),     // StatusOr<int>
  rpl::UnwrapStatusOr(),         // int
  rpl::Transform(&Squared),      // int
  rpl::To<std::vector>()         // vector<int>
);                                     // StatusOr<vector<int>>
```

Monadic StatusOr

```
absl::StatusOr<int> ParseInt(std::string_view);
int Squared(int i);

absl::StatusOr<std::vector<std::string>> values = ...;

absl::StatusOr<std::vector<int>> result = rpl::Apply(
  values,                              // StatusOr<vector<string>>
  rpl::UnwrapStatusOrComplete(), // std::vector
  rpl::Transform(&ParseInt),           // StatusOr<int>
  rpl::UnwrapStatusOr(),               // int
  rpl::Transform(&Squared),            // int
  rpl::To<std::vector>()               // vector<int>
);                                     // StatusOr<vector<int>>
```

Monadic StatusOr

```
absl::StatusOr<int> ParseInt(std::string_view);
int Squared(int i);

absl::StatusOr<std::vector<std::string>> values = ...;

absl::StatusOr<std::vector<int>> result = rpl::Apply(
  values,                              // StatusOr<vector<string>>
  rpl::UnwrapStatusOrComplete(), // std::vector
  rpl::Transform(&ParseInt),     // StatusOr<int>
  rpl::UnwrapStatusOr(),         // int
  rpl::Transform(&Squared),      // int
  rpl::To<std::vector>()         // vector<int>
);                                     // StatusOr<vector<int>>
```

Monadic StatusOr

```
absl::StatusOr<int> ParseInt(std::string_view);
int Squared(int i);

absl::StatusOr<std::vector<std::string>> values = ...;

absl::StatusOr<std::vector<int>> result = rpl::Apply(
  values,                              // StatusOr<vector<string>>
  rpl::UnwrapStatusOrComplete(), // std::vector
  rpl::Transform(&ParseInt),     // StatusOr<int>
  rpl::UnwrapStatusOr(),         // int
  rpl::Transform(&Squared),      // int
  rpl::To<std::vector>()         // vector<int>
);                                     // StatusOr<vector<int>>
```

Monadic StatusOr

```
absl::StatusOr<int> ParseInt(std::string_view);
int Squared(int i);

absl::StatusOr<std::vector<std::string>> values = ...;

absl::StatusOr<std::vector<int>> result = rpl::Apply(
  values,                              // StatusOr<vector<string>>
  rpl::UnwrapStatusOrComplete(), // std::vector
  rpl::Transform(&ParseInt),     // StatusOr<int>
  rpl::UnwrapStatusOr(),         // int
  rpl::Transform(&Squared),      // int
  rpl::To<std::vector>()         // vector<int>
);                                     // StatusOr<vector<int>>
```

Monadic StatusOr

```
absl::StatusOr<int> ParseInt(std::string_view);
int Squared(int i);

absl::StatusOr<std::vector<std::string>> values = ...;

absl::StatusOr<std::vector<int>> result = rpl::Apply(
  values,                              // StatusOr<vector<string>>
  rpl::UnwrapStatusOrComplete(), // std::vector
  rpl::Transform(&ParseInt),     // StatusOr<int>
  rpl::UnwrapStatusOr(),         // int
  rpl::Transform(&Squared),      // int
  rpl::To<std::vector>()         // vector<int>
);                                     // StatusOr<vector<int>>
```

Monadic StatusOr

```
absl::StatusOr<int> ParseInt(std::string_view);
int Squared(int i);

absl::StatusOr<std::vector<std::string>> values = ...;

absl::StatusOr<std::vector<int>> result = rpl::Apply(
  values,                              // StatusOr<vector<string>>
  rpl::UnwrapStatusOrComplete(), // std::vector
  rpl::Transform(&ParseInt),     // StatusOr<int>
  rpl::UnwrapStatusOr(),         // int
  rpl::Transform(&Squared),      // int
  rpl::To<std::vector>()         // vector<int>
);                                     // StatusOr<vector<int>>
```

HIGHER-ORDER STAGES

Tee splits outputs to multiple stages

```
std::vector<int> values = {...};
auto [min, max, count] = rpl::Apply(
  values,
  rpl::Tee(
    rpl::Min(), rpl::Max(), rpl::Count()));
```

Tee splits outputs to multiple stages

```
std::vector<int> values = {...};
auto [min, max, count] = rpl::Apply(
  values,
  rpl::Tee(
    rpl::Min(), rpl::Max(), rpl::Count()));
```

Group elements

```
struct Employee {
  int id;
  bool is_fulltime;
  std::string org;
};
std::vector<Employee> employees = {...};
auto counts = rpl::Apply(
  employees,
  rpl::MapGroupBy<std::unordered_map>(
    &Employee::org
    rpl::Filter(&Employee::is_fulltime),
    rpl::Count()),
  rpl::MakePair(),
  rpl::To<std::vector>());
```

Group elements

```
struct Employee {
  int id;
  bool is_fulltime;
  std::string org;
};
std::vector<Employee> employees = {...};
auto counts = rpl::Apply(
  employees,
  rpl::MapGroupBy<std::unordered_map>(
    &Employee::org
    rpl::Filter(&Employee::is_fulltime),
    rpl::Count()),
  rpl::MakePair(),
  rpl::To<std::vector>());
```

REFERENCE INPUTS

Const reference iteration by default

```
std::vector<std::unique_ptr<int>> values = {...};
rpl::Apply(
  values,  // values not copied!
  rpl::ForEach(
    [](auto&& ptr) {
    static_assert(std::is_same_v<
      decltype(ptr),
      const std::unique_ptr<int>&
    >);
  }));
```

Const reference iteration by default

```
std::vector<std::unique_ptr<int>> values = {...};
rpl::Apply(
  values,  // values not copied!
  rpl::ForEach(
    [](auto&& ptr) {
    static_assert(std::is_same_v<
      decltype(ptr),
      const std::unique_ptr<int>&
    >);
  }));
```

Const reference iteration by default

```
std::vector<std::unique_ptr<int>> values = {...};
rpl::Apply(
  values,  // values not copied!
  rpl::ForEach(
    [](auto&& ptr) {
    static_assert(std::is_same_v<
      decltype(ptr),
      const std::unique_ptr<int>&
    >);
  }));
```

Mutable references with `std::ref`

```
rpl::Apply(
  std::ref(values),
  rpl::ForEach(
    [](auto&& ptr) {
      static_assert(std::is_same_v<
        decltype(ptr),
        std::unique_ptr<int>&
      >);
  }));
```

Mutable references with `std::ref`

```
rpl::Apply(
  std::ref(values),
  rpl::ForEach(
    [](auto&& ptr) {
      static_assert(std::is_same_v<
        decltype(ptr),
        std::unique_ptr<int>&
      >);
  }));
```

Mutable references with `std::ref`

```
rpl::Apply(
  std::ref(values),
  rpl::ForEach(
    [](auto&& ptr) {
      static_assert(std::is_same_v<
        decltype(ptr),
        std::unique_ptr<int>&
      >);
  }));
```

R-Values when moved

```
rpl::Apply(
  std::move(values),
  rpl::ForEach(
    [](auto&& ptr) {
      static_assert(std::is_same_v<
        decltype(first),
        std::unique_ptr<int>&&
      >);
  }));
```

R-Values when moved

```
rpl::Apply(
  std::move(values),
  rpl::ForEach(
    [](auto&& ptr) {
      static_assert(std::is_same_v<
        decltype(first),
        std::unique_ptr<int>&&
      >);
  }));
```

R-Values when moved

```
rpl::Apply(
  std::move(values),
  rpl::ForEach(
    [](auto&& ptr) {
      static_assert(std::is_same_v<
        decltype(first),
        std::unique_ptr<int>&&
      >);
  }));
```

CONVENIENCES

Initializing `std::vector` with `unique_ptr`

```
// Doesn't work
std::vector v{make_unique<int>(0), make_unique<int>(1)};

// Works
auto v = rpl::Apply(
  {std::make_unique<int>(0), std::make_unique<int>(1)},
  rpl::To<std::vector>());
```

Moving keys out of map or set

```
// Doesn't work
std::vector<std::string> v(std::move(s).begin(),
  std::move(s).end());

// Doesn't work
std::vector<std::string> v2(std::make_move_iterator(s.begin()),
  std::make_move_iterator(m.end()));

// Works
auto v = rpl::Apply(std::move(s), rpl::To<std::vector>());
```

programming pearls

by Jon Bentley
with Special Guest Oysters
Don Knuth and Doug McIlroy

A LITERATE PROGRAM

CHALLENGE

Given a text file and an integer k, print the k most common words in the file (and the number of their occurrences) in decreasing frequency.

STAGES

READLINES

```
auto ReadLines() {
  return rpl::TransformComplete([](std::istream &is) {
    return rpl::Generator<absl::string_view>(
        [&is, line = std::string()](auto output) mutable {
          if (std::getline(is, line)) {
            std::move(output)(line);
          }
        });
  });
}
```

SPLIT STRING

```
auto SplitString() {
 return rpl::Compose(
   rpl::Transform(
    [](absl::string_view s) {
      return absl::StrSplit(
        s,
        absl::ByAsciiWhitespace(),
        absl::SkipEmpty()
      );
   }),
   rpl::Flatten()
  );
}
```

LOWER CASE STRING

```
auto LowerCaseString() {
  return rpl::Transform([](absl::string_view s) {
    return rpl::Apply(
        s,
        rpl::Transform([](char c) { return std::tolower(c); }),
        rpl::To<std::string>()
    );
  });
}
```

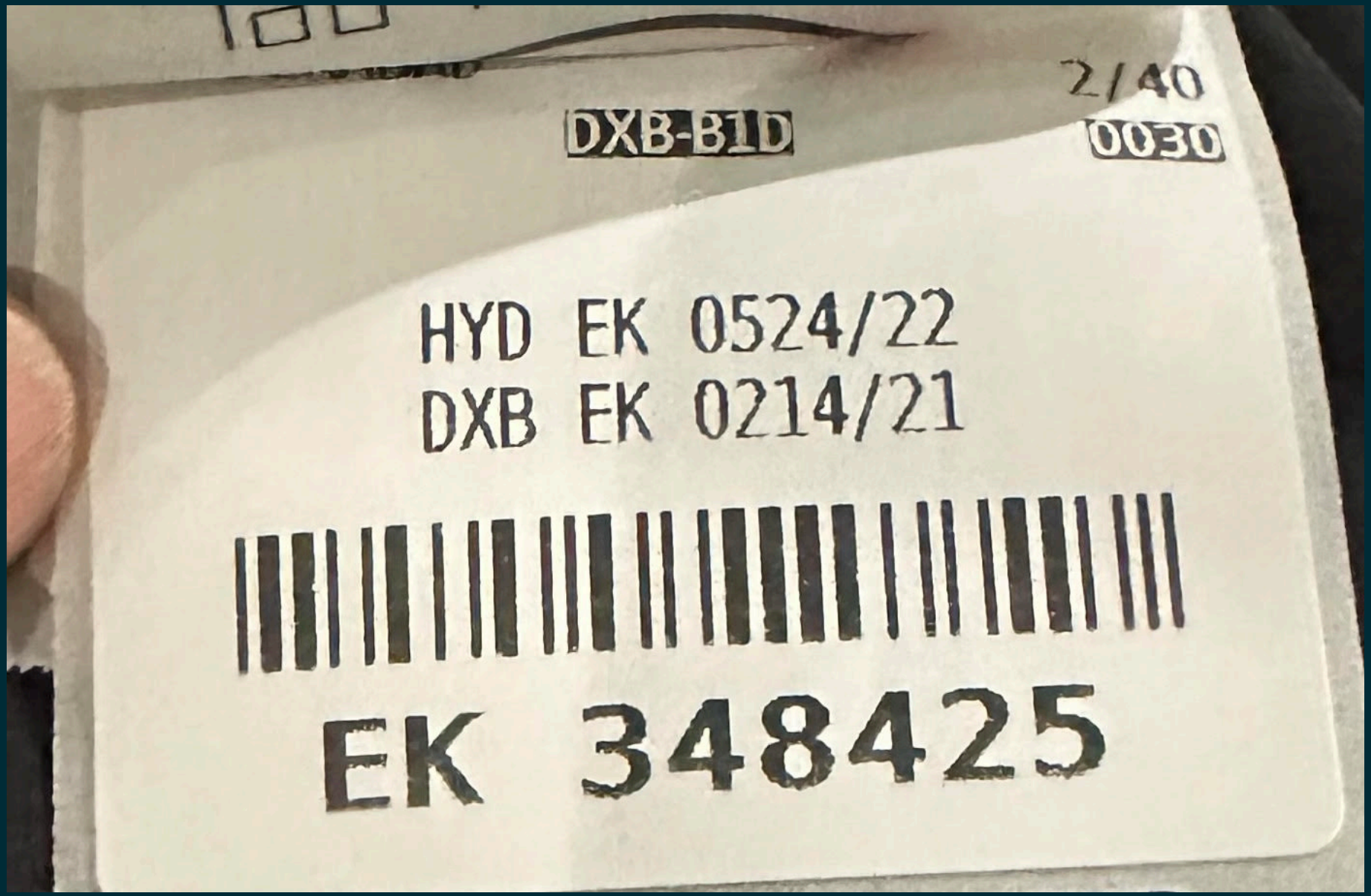
UNIQUE COUNTS

```
auto UniqueCounts() {
 return rpl::Compose(
  rpl::MapGroupBy<absl::flat_hash_map>(
    std::identity(), rpl::Count()),
  rpl::Swizzle<1, 0>(),
  rpl::MakePair()
 );
}
```

WORD COUNTS PIPELINE

```
rpl::Apply(
 std::ref(std::cin),
 ReadLines(),
 SplitString(),
 LowerCaseString(),
 UniqueCounts(),
 rpl::To<std::vector>(),
 rpl::PartialSort(k, std::greater<>()),
 rpl::Take(k),
 rpl::ExpandTuple(),
 rpl::ForEach([](int n, const auto &s) {
   std::cout << n << " " << s;
 })
);
```


SINGAPORE AIRLINES


DXB-BID
2/40
0030
HYD EK 0524/22
DXB EK 0214/21
EK 348425

QUESTIONS