RANGES: DESIGN AND PITFALLS

- **Ranges: the STL to the Next Level**
- **The Terrible Problem Of Incrementing A Smart Iterator**
- **The Surprising Limitations of C++ Ranges Beyond Trivial Cases**
- **An Alternative Design to Iterators and Ranges, Using std::optional**

Ranges: the STL to the Next Level

As seen in a dedicated post, The C++ Standard Template Library (STL) is a fantastic tool for making code more correct and expressive. It is mainly composed of two parts:

- The **containers**, such as `std::vector` or `std::map` for instance,
- The **algorithms**, a fairly large collection of generic functions that operate amongst others on containers. They are mostly found under the `algorithm` header.

A lot of manual operations performed on containers with for loops can be replaced by calls to algorithms of the STL. This has the effect of making the code clearer, because instead of having to mentally parse a complex for loop, a reader of the code can instantly understand what is going on if the offending for loops are replaced with explicit names such as `std::copy`, `std::partition` or `std::rotate`.

However, the STL has several aspects that can be improved. In this post we focus on two of them:

- All algorithms manipulate **iterators** pointing into the collection they operate on. While this is handy in specific cases like stopping at a precise point in a container, the largely general case is to traverse the whole container from its .begin() to its .end().Therefore, portions of code that use the STL end up being littered with iterators:

```
std::copy(v1.begin(), v1.end(), std::back_inserter(v2));
std::set_difference(v2.begin(), v2.end(), v3.begin(),
v3.end(), std::back_inserter(v4));
std::transform(v3.begin(), v3.end(), std::back_inserter(v4));
```

(Note: the std::back_inserter used above is an output iterator that does a push_back into the container it is passed, every time it is assigned to. This relieves the programmer from the sizing of the output)

- Algorithms **do not compose well**. I found that a recurring need encountered by C++ developers who use the STL is to apply a function only on elements of a

collection that satisfy a predicate.Applying a function f on all the elements of a collection input and putting the results in a vector output is achieved by `std::transform`:

```
std::transform(input.begin(), input.end(),
std::back_inserter(output), f);
```

And filtering the elements on a predicate p is done with `std::copy_if`:

```
std::copy_if(input.begin(), input.end(), std::back_inserter(output),
p);
```

But there is no easy way to combine these two calls, an

d there is no such thing as a “transform_if” algorithm.

Ranges provide a different approach to the STL that solves these two issues in a very elegant manner. Ranges were initially introduced in Boost, and are now on their way to standardization. I believe they will have a major impact on the way we deal with collections in code.

The concept of Range

At the center of all this is the concept of **Range**. Essentially, a range is something that can be **traversed**. More precisely, a range is something that has a `begin()` and an `end()` method, that return objects (iterators) that let you iterate over the range (that is, move along the elements of the range, and be dereferenced to access these elements).

Expressed in pseudocode a range would be something that complies to the following interface:

```
Range
{
    begin()
    end()
}
```

In particular, this implies that **all STL containers are themselves ranges**.

Ranges were already used in some way by code using the STL before the Range concept was defined, but clumsily. As seen at the beginning of this post they were manipulated directly with two iterators, typically a begin and an end. With ranges though, you generally don't see iterators. They are here, but abstracted away by the concept of range.

This is important to understand. Iterators are technical constructs that let you iterate over a collection, but they are generally too technical for your functional code. Most of the time, what you are really trying to represent is a range, which corresponds better to the level of abstraction of your code. Like a range of cash flows, a range of lines in a screen, or a range of entries coming up from the database.

So coding in terms of ranges is a huge improvement, because in that sense iterators violate the principle of respecting levels of abstraction, which I deem is the most important principle for designing good code.

In range libraries, STL algorithms are redefined to take directly ranges as parameters, instead of two iterators, like:

```
ranges::transform(input, std::back_inserter(output), f);
```

As opposed to:

```
std::transform(input.begin(), input.end(),
std::back_inserter(output), f);
```

Such algorithms reuse the STL versions in their implementation, by forwarding the begin and the end of the range to the native STL versions.

Smart iterators

Even though they are abstracted away by ranges, range traversals are implemented with iterators. The full power of ranges comes from its combination with smart iterators. Generally speaking, an iterator of a collection has two responsibilities:

- Moving along the elements of the collection (++, –, etc.)

- Accessing the elements of the collection (*, ->)

For example, a vector iterator does just this. But “smart” iterators that originated in boost customize one or both of these behaviours. For instance:

- The `transform_iterator` is constructed with another iterator `it` and a function (or function object) `f`, and customizes the way it accesses elements: when dereferenced, the `transform_iterator` applies `f` to `*it` and returns the result.
- The `filter_iterator` is constructed with another iterator `it`and a predicate `p`. It customizes the way its moves: when advancing by one (++) a `filter_iterator`, it advances its underlying iterator `it` until it reaches an element that satisfies the predicate or the end of the collection.

Combining Ranges and Smart iterators: Range adaptors

The full power of ranges comes with their association with smart iterators. This is done with **range adaptors**.

A range adaptor is an object that can be combined with a range in order to produce a new range. A subpart of them are **view adaptors**: with them, the initial adapted range remains unchanged, while the produced range does not contain elements because it is rather a view over the initial one, but with a customized iteration behaviour.

To illustrate this let’s take the example of the **`view::transform`** adaptor. This adaptor is initialized with a function and can be combined with a range to produce a view over it, that has the iteration behaviour of a `transform_iterator` over that range. Range adaptors can be combined with ranges with `operator|`, which gives them a elegant syntax.

With the following collection of numbers:

```
std::vector numbers = { 1, 2, 3, 4, 5 };
```

The range

```
auto range = numbers | view::transform(multiplyBy2);
```

is a view over the vector numbers that has the iteration behaviour of a `transform_iterator` with the function `multiplyBy2`. So when you iterate over this view, the results you get are all these numbers, multiplied by 2. For instance:

```
ranges::accumulate(numbers | view::transform(multiplyBy2), 0);
```

returns `1*2 + 2*2 + 3*2 + 4*2 + 5*2 = 30` (similarly to `std::accumulate`, `ranges::accumulate` does the sum of the elements of the range it is passed to).

There are plenty of other range adaptors. For example, **`view::filter`** takes a predicate and can be combined with a range to build a view over it with the behaviour of a `filter_iterator`:

```
ranges::accumulate(numbers | view::filter(isEven), 0);
```

returns `2 + 4 = 6`.

An important thing to note is that the ranges resulting from associations with range adaptors, although they are merely view over the ranges they adapt and don't actually store elements, answer to the range interface (begin, end) so they are **themselves ranges**. Therefore adaptors can adapt ad apted ranges, and can be effectively combined the following way:

```
ranges::accumulate(numbers | view::filter(isEven) |
view::transform(multiplyBy2), 0);
```

returns `2*2 + 4*2 = 12`. And this gives a solution to the initial problem of not being able to combine algorithms together.

Conclusion

Ranges raise the level of abstraction of code using the STL, therefore clearing up code using the STL from superfluous iterators. Range adaptors are a very powerful and expressive tool to apply operations on elements of a collection, in a modular way.

Ranges are the future of the STL. To go further you can have a look at the initial range library in boost or to the proposal for standardization from Eric Niebler. As this proposal depends on concepts, that were not included in C++17, ranges have not been standardized yet. Until they are, you can dig into Eric Niebler's range library range-v3 that is compatible with the current versions of the C++ language. It is available in Visual Studio 2015 Update 3 with a fork of the popular range-v3 library.

Understand ranges better with the new Cartesian Product adaptor

A couple of days ago, the range-v3 library got a new component: the `view::cartesian_product` adaptor.

Understanding what this component does, and the thought process that went through its creation is easy and will let you have a better grasp of the range library. (Note that you could just as well understand all the following by looking at the `zip` adaptor. But cartesian_product is brand new, so let's discover this one, in order to hit two birds with one stone.)

Oh maybe you're wondering why you would need to understand the range library?

Like I explained in details on Arne Mertz's blog Simplify C++!, ranges are the future of the STL. Essentially, the STL is a powerful tool for writing expressive code, and ranges are a very well designed library that takes it much farther. Ranges are expected to be included in the next C++ standard, hopefully C++20, and until then they are available to test on Eric Niebler's github, its author. So in a nutshell, you want to learn ranges to understand where the craft of writing expressive C++ is heading to.

Motivation

The purpose of the cartesian_product adaptor is to iterater over all the possible combinations of the elements of several collections.

We will use toy examples in this articles to keep all the business specific aspects away, but an example of where this can be useful is where objects have versions. In such a case you may want to generate all possible objects for all possible dates for example.

But for our purpose we will use the following 3 collections. First a collection of numbers:

```
std::vector<int> numbers = {3, 5, 12, 2, 7};
```

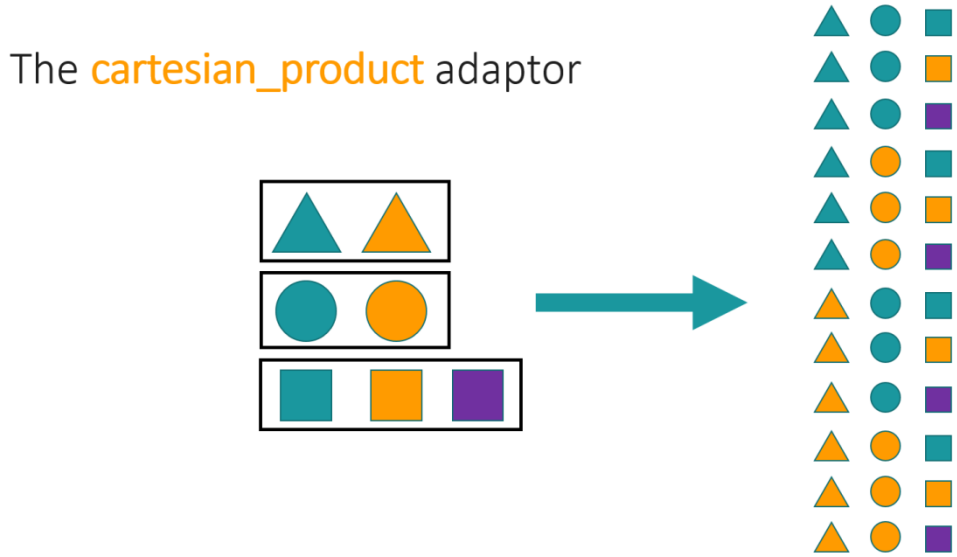
then a collection of types of food that are typically served at a meetup, represented by strings:

```
std::vector<std::string> dishes = {"pizzas", "beers", "chips"};
```

and finally a collection of places, also represented by strings for simplicity:

```
std::vector<std::string> places = {"London", "Paris", "NYC",
"Berlin"};
```

Now we want to do an action, like printing a sentence, with every possible combination of the elements of these 3 collections.



Putting the behaviour into an algorithm

Here was my first attempt at writing a generic function that could apply a function over all the possible combinations of several collections. I am purposefully taking away all the variadic aspects here, in order to keep the focus on the responsibilities of the algorithms:

```
template<typename Collection1, typename Collection2, typename
Collection3, typename Function>
void cartesian_product(Collection1&& collection1, Collection2&&
collection2, Collection3&& collection3, Function func)
{
    for (auto& element1 : collection1)
        for (auto& element2 : collection2)
            for (auto& element3 : collection3)
                func(element1, element2, element3);
}
```

And this does the job. Indeed, the following call:

```
cartesian_product(numbers, dishes, places,
    [](int number, std::string const& dish, std::string const&
place)
    { std::cout << "I took " << number << ' ' << dish << " in " <<
place << ".\n";});
```

outputs this:

I took 3 pizzas in London.

I took 3 pizzas in Paris.

I took 3 pizzas in NYC.

I took 3 pizzas in Berlin.

I took 3 beers in London.

I took 3 beers in Paris.

I took 3 beers in NYC.

I took 3 beers in Berlin.

I took 3 chips in London.

I took 3 chips in Paris.

I took 3 chips in NYC.

I took 3 chips in Berlin.

I took 5 pizzas in London.

I took 5 pizzas in Paris.

I took 5 pizzas in NYC.

I took 5 pizzas in Berlin.

I took 5 beers in London.

I took 5 beers in Paris.

I took 5 beers in NYC.

I took 5 beers in Berlin.

I took 5 chips in London.

I took 5 chips in Paris.

I took 5 chips in NYC.

I took 5 chips in Berlin.

The limits of an algorithm

It looks ok, but the above code stops working if I slightly change the requirement. Say now that we no longer want a function to directly write to the console. To decouple the code from the IO, we want to output the various combinations into a container of strings.

And then we're stuck with the above implementation, because it doesn't return anything. (If it crossed your mind to store the output in the function by making it a function object, then you must be under an amount of stress which is higher than necessary. To relax, I suggest you read STL function objects: Stateless is Stressless).

In fact, the above algorithm is sort of the equivalent of `std::for_each` for all possible combinations, because it iterates over all of them and applies a function. And what we would need here is rather an equivalent of `std::transform` (more about this central algorithm here).

Are we to recode a new cartesian_product that takes an output collection and a function, like `std::transform`? It feels wrong, doesn't it?. We'd rather **take the iterating responsibility out of the algorithms**. And this is exactly what the cartesian_product adaptor does for you.

The cartesian_product adaptor constructs a view over a set of collections, representing it as a range of tuples containing every possible combinations of the elements in the collections. Then the function has to take a tuple containing its arguments. Note that it would be preferable to keep taking the arguments directly instead of through a tuple, but more on this later.

Here is an example to satisfy the need of outputing the sentences into a string container:

```
std::string meetupRecap(std::tuple<int, std::string, std::string>
const& args)
{
    int number = std::get<0>(args);
    std::string const& dish = std::get<1>(args);
    std::string const& place = std::get<2>(args);

    std::ostringstream result;
    result << "I took " << number << ' ' << dish << " in " << place
<< '.';
    return result.str();
}

std::vector<std::string> results;
transform(ranges::view::cartesian_product(numbers, dishes, places),
std::back_inserter(results), meetupRecap);
```

And the **same adaptor** can also be used to perform the output to the console, without having to write a specific algorithm:

```
void meetupRecapToConsole(std::tuple<int, std::string, std::string>
const& args)
{
    int number = std::get<0>(args);
```

```
    std::string const& dish = std::get<1>(args);
    std::string const& place = std::get<2>(args);


    std::cout << "I took " << number << ' ' << dish << " in " <<
place << ".\n";
}

for_each(ranges::view::cartesian_product(numbers, dishes, places),
meetupRecapToConsole);
```

This adaptor effectively takes the responsability of generating all the possible combinations of elements, thus letting us reuse regular algorithms, such as `for_each` and `transform`.

The official cartesian_product range adaptor

A couple of months ago I came up with this adaptor and proposed it to Eric Niebler:

Eric responded positively and a few weeks later, Casey Carter implemented it inside the range-v3 library (thanks Casey!):

...which is how range-v3 got this new adaptor.

To me it's a good addition, and I think that the interface using tuples can be further improved. There is a way to encapsulate the tuple machinery into another component – but we will get into this topic in another post, another time.

The Terrible Problem Of Incrementing A Smart Iterator

The Terrible Problem Of Incrementing A Smart Iterator (or TPOIASI) is a difficulty that arises when implementing smart iterators.

But even if you don't implement smart iterators, you may use them in a disguised form, now or in the future. And then, the TPOIASI might impact your code in a subtle manner.

Since the world is moving towards smart iterators – well, at least the C++ world – you should know what the TPOIASI is about, because it may try to bite you some day.

Smart iterators 101

To understand the TPOIASI, let's start by its last two letters: the Smart Iterators. If you're already familiar with smart iterators and range adaptors, you can skip to the next section.

Iterators

An iterator is a component linked to a range of objects (for example, to an STL container like `std::vector`), that has two missions:

- giving access to the objects in the range, with `operator*`
- moving along the range, with `operator++`, to access all the elements in the range successively.

Most of the STL iterators, such as those of `std::vector` or `std::map`, fulfil these two roles, that together allow to traverse a collection.

Smart iterators

This is not an official term, but a **smart** iterator is an iterator, so it also does those two jobs. But it does them in a special way.

One example of a smart iterator is the **transform iterator**, that does not just give access to an element of a range with its `operator*`. Instead, it gives the result of applying a function `f` to the element of the range.

Another example is the **filter iterator**. Its `operator++` does not just moves to the adjacent element in the range. It moves to the next element in the range that satisfies a predicate `p`, (potentially moving past several elements of the range that wouldn't satisfy `p`).

Another important aspect of smart iterators is that they can **combine** with other iterators. For example, a transform iterator can be plugged onto a vector iterator. In its `operator*`, the transform iterator calls the `operator*` of the vector iterator, and applies `f` on the value that the latter returns.

We could then have a filter iterator plugged onto a transform iterator, itself plugged onto a vector iterator. The result is an iterator that skips some of the results of applying `f` to the vector elements, if they don't satisfy `p`. And smart iterators can combine into arbitrarily long chains.

Range adaptors

When the STL manipulates two iterators, like in its algorithms, it's often to represent a range: one iterator represents the beginning of a range, and the other the end. Rather than having to manipulate those two iterators, it's often more convenient to directly use a **range** instead.

A simple definition of a range is: something that provides a `begin()` and an `end()` iterator. In this definition, STL containers are ranges.

But the simplest implementation of a range is a structure that contains two iterators and offers a `begin()` and `end()` interface that return them.

Back to our smart iterators now. If we have two smart iterators, like two transform iterators, plugged onto the begin and end of the same vector, it can then define a smart range: a range

that, when you iterate over it, gives you the results of applying `f` to each element of the vector.

Packaging this feature nicely into a component that will do the job of generating transform iterators for you, gets to something like this:

```
myVector | transform([](int n){ return n * 2; });
```

This a view over `myVector`, where you see all its values multiplied by 2. This is the sort of code you can write by using ranges libraries, such as range-v3. And ranges may well be the future of the STL.

And combined with filter iterators:

```
myVector | transform([](int n){ return n * 2; })
         | filter([](int n){ return n % 4; });
```

This is a view of the values of `myVector` multiplied by 2, that can be divided by 4.

Now that we have a better feel on what Smart Iterators are, let's move on to the Terrible Problem Of Incrementing A Smart Iterator.

The TPOIASI

To illustrate the issue, let's build a simple example using a range library. Here I'm using range-v3 which is available on Wandbox:

```
// Input vector
std::vector<int> numbers = {1, 2, 3, 4, 5};

// Output vector
std::vector<int> results;

//Apply transform and filter
ranges::push_back(results,
                  numbers | ranges::view::transform(times2)
                          | ranges::view::filter(isMultipleOf4));

// Display results
for (auto result : results)
{
    std::cout << result << ' ';
```

```
}
```

With times2 and isMultipleOf4 being:

```
int times2(int n)
{
   return n * 2;
}

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

Here is what the code outputs:

4 8

Indeed, the `numbers` piped into `transform` give `{2, 4, 6, 8, 10}`, and the multiples of 4 here are `4` and `8`, so it's all fine.

Except there is a problem with this code, and a subtle one because it doesn't show when you look at the code. Let's trace the calls to the function in the `transform` adaptor:

```
int times2(int n)
{
   std::cout << "transform " << n << '\n';
   return n * 2;
}
```

Now here is what the code outputs:

transform 1

transform 2

transform 2

transform 3

transform 4

transform 4

transform 5

4 8

For some values the function is called several times!

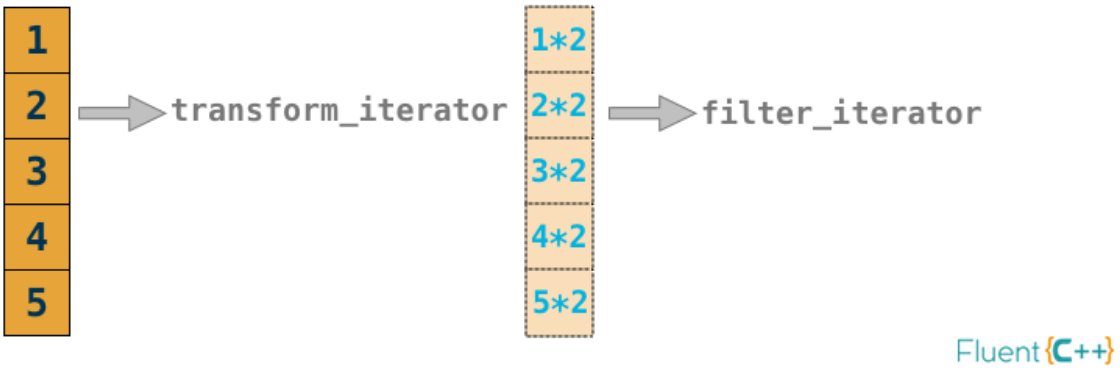
This may not matter, like in our example with `int`. But if the function was doing a big computation, then we would notice a performance impact (it happened to me once). Or in the (questionable) case where the function has side effects, we would probably have wrong results.

Now why does the library call the function several times in the first place? To understand this, we need to think about how to implement a filter iterator.

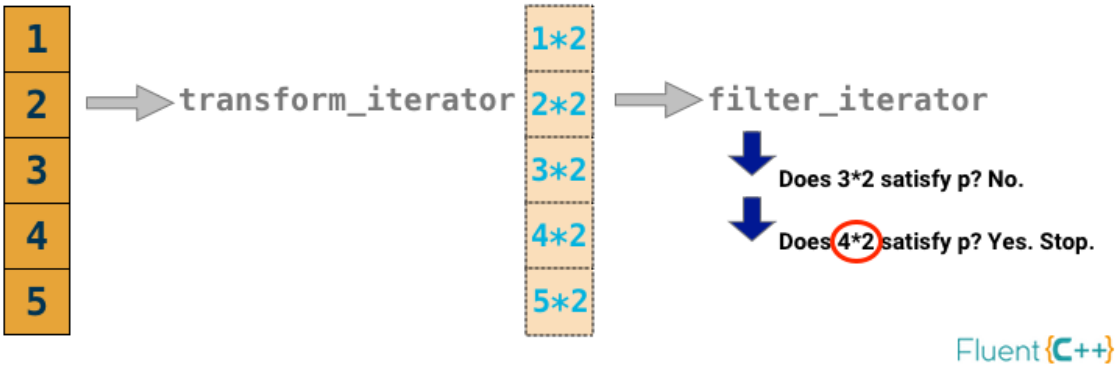
The cause of the TPOISI

The cause of the TPOISI lies in its central I: the Increment operator, `operator++`, and more specifically the one of the filter iterator.

How would you implement the `operator++` of the filter iterator? Imagine that your filter iterator is sitting somewhere in the collection, for instance in front of the first element that satisfies the predicate. In our example, that would be 2*2 = 4:

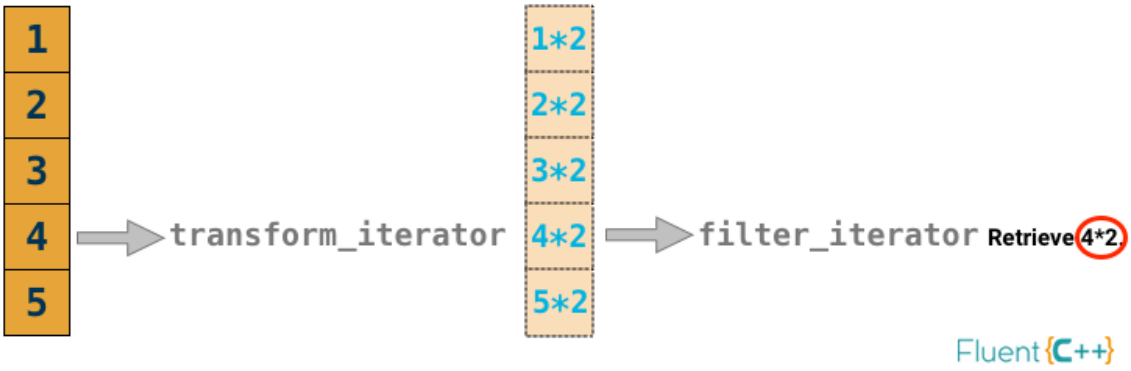


So let's call `operator++` on the filter iterator. The filter iterator calls `operator++` on its underlying iterator (here, the transform iterator) and has to **peek** to the element to check where to stop:



But the filter iterator makes its check on the value returned by the transform iterator. And the transform iterator provides its value by applying its function. So here, we have a our function applied to `3` once and then applied to `4` once.

After calling `operator++`, the next step to traverse the collection is to get a value from filter iterator, by calling `operator*` This is what `std::copy` does, for example. And to provide a value, the filter iterator asks it to its underlying transform iterator, which then calls the function a second time on `4` to compute 4*2:



This is why the function `times2` is called twice on `4`.

How to work around the issue?

Let's finish up with the first letters of the TPOIASI, the ones that makes it a Terrible Problem.

I call it that because it seems to me like a structural problem in the filter iterator, and filtering is a common need among the manipulations on ranges. Note that the transform iterator doesn't suffer from the TPOIASI: with a transform on a transform, none of them gets called more than once.

So what's special about the filter iterator? It is that it customizes the iteration on the underlying container, and has to peek to the underlying iterator to do that.

The issue can be reproduced in range-v3, I had also encountered it when trying to implement a filter iterator, and can't see how to fix it. If you see how, please write a comment.

It's not a showstopper for ranges, but it can be a real issue for some cases. In all cases, it's good to be aware of it.

However, a couple of weeks ago, we have seen another approach to chain up operations on collections: smart output iterators, which are a sort of symmetry to the approach of ranges. Smart output iterator don't suffer from the The Terrible Problem Of Incrementing A Smart Iterator, or at least not as much as ranges. Even though they have to make a little sacrifice.

How do smart output iterators offer resistance to the TPOIASI? What sacrifice will they have to make? This is what is coming up in the next post on Fluent C++.

The Surprising Limitations of C++ Ranges Beyond Trivial Cases

Today we have a guest post from Alex Astashyn. Alex is a tech lead for the RefSeq resource at the National Center for Biotechnology Information.

Note: The opinions expressed in this article are those of the author. Also, I can't count myself as a "range expert", so some of the information pertaining to ranges might be factually incorrect (leave a comment if you spot anything egregiously wrong).

In this article I discuss the problems and limitations I've encountered with c++ ranges.

I also introduce my own library, rangeless that distills all functionality I expected to have been fulfilled by ranges. It allowed me to tackle much more expanded scope of interesting applicable real-life use-cases.

Prologue

Like any fan of functional-oriented declarative stateless programming, I thought ranges looked very promising. However, trying to use them in practice proved to be a very frustrating experience.

I kept trying to write what seemed to me like perfectly reasonable code, yet the compiler kept barfing pages of error messages I could not make sense of. Eventually I realized the error of my ways. I thought of ranges like of UNIX pipelines `cat file | grep ... | sed ... | sort | uniq -c | sort -nr | head -n10`, but that is not so...

Examples

Example 1: Intersperse

Let's try writing a view that intersperses a delimiter between input elements.

(This functionality is provided by range-v3, so we can compare and contrast the approaches)

```
// inputs:    [x1, x2, ... xn]
        // transform: [[x1, d], [x2, d], ... [xn, d]]
        // flatten:   [ x1, d, x2, d, ... xn, d ]
        // drop last: [ x1, d, x2, d, ... xn ]
        auto intersperse_view =
        view::transform([delim](auto inp)
        {
            return std::array<decltype(inp), 2>{{ std::move(inp),
delim }};
        })
      | view::join // also called concat or flatten in functional
languages
      | view::drop_last(1); // drop trailing delim
```

The `transform | join` composition above is a common operation on streams that transforms each input into a sequence of outputs, and flattens the resulting sequence-of-sequences.

```
[x] -> (x -> [y]) -> [y]
```

Some languages have a separate abstraction for this, e.g. `flat_map` in Elixir or `SelectMany` in LINQ.

Adhering to Principle of Least Astonishment, it seems like the above ought to work. (if you have not seen this talk, I cannot recommend it enough).

However, this will not compile with `range-v3`. What gives? It turns out the problem is that `view::join` doesn't like the fact that the subrange (returned collection) is a container returned as rvalue. I came up with the following hack: views (sometimes) compose with rvalues of views, so let's wrap the container return-value as a view!

```
view::transform([delim](auto inp)
        {
            return view::generate_n([delim, inp, i = 0]() mutable
            {
                return (i++ == 0) ? inp : delim;
            }, 2);
        })
```

Or, generalizing, if we want to return a container, e.g. a vector, as a view in some other use-case:

```
view::transform([](int x)

    {
        auto vec = ... ;
        return view::generate_n([i = 0, vec = std::move(vec)]() mutable
        {
            return std::move(vec[i++]);
        }, vec.size());
    })
  | view::join // now join composes with transform
```

Isn't this clever? Maybe, but having to come up with clever hacks to be able to do something as basic as this is not a good sign.

It turns out I was not the first person to hit this problem. The library implementers presented their own workarounds. As noted by Eric Niebler here, my solution is "illegal" because by capturing the vector in the view no longer satisfies O(1) copy complexity requirement.

That said, if we peek under the hood of `view::generate` or `view::generate_n` we'll see that they cache the last generated value, so having `view::generate` yield a `std::string`, or `std::vector`, or a type containing those, you're not satisfying the library requirements already.

Are we done with the example? Almost.

We have:

```
...
| view::join
| view::drop_last(1);
```

You'd think that `drop_last` would internally keep a queue of `n` elements in a circular buffer and would simply discard it upon reaching last input. `range-v3` views, however may not buffer elements, so `view::drop_last` has to impose `SizedRange` or `ForwardRange` requirement on the input, whereas `view::join` returns an `InputRange` (even if it receives a `ForwardRange` as input). This kills not only the composition, or any hope of lazy evaluation (you have to eagerly dump your entire `InputRange` (hopefully finite) to a `std::vector` first to convert it to a `ForwardRange`).

So how would we implement this? We'll get to it later...

Example 2:

Below is an example implemented with `rangeless` library (a slightly modified version of Knuth-vs-McIlroy challenge to make it a little more interesting).

```
namespace fn = rangeless::fn;
    using fn::operators::operator%;
    //
    // Top-5 most frequent words from stream chosen among the words
of the same length.
    //
    auto my_isalnum = [](const int ch)
    {
        return std::isalnum(ch) || ch == '_';
    };
    fn::from( // (1)
        std::istreambuf_iterator<char>(std::cin.rdbuf()),
        std::istreambuf_iterator<char>{ /* end */ })
      % fn::transform([](const char ch) // (2)
        {
            return std::tolower(uint8_t(ch));
        })
      % fn::group_adjacent_by(my_isalnum) // (3)
        // (4) build word->count map
      % fn::foldl_d([&](std::map<std::string, size_t> out, const
std::string& w)
        {
            if(my_isalnum(w.front())) {
                ++out[ w ];
            }
            return out; // NB: no copies of the map are made
                                // because it is passed back by
move.
        })
      % fn::group_all_by([](const auto& kv) // (5) kv is (word,
count)
        {
            return kv.first.size(); // by word-size
        })
      % fn::transform( // (6)
            fn::take_top_n_by(5UL, fn::by::second{})) // by count
      % fn::concat() // (7) Note: concat is called _join_ in range-
v3
      % fn::for_each([](const auto& kv)
        {
            std::cerr << kv.first << "\t" << kv.second << "\n";
        })
      ;
```

As you can see, the code is very similar to ranges in style, but the way it works under the hood is `entirely different` (will be discussed later).

Trying to rewrite this with `range-v3` we would encounter the following problems:

- (3) This will not work because `view::group_by` requires a `ForwardRange` or stronger.
- (4) How does one do a composeable left-fold (one of the three pillars of filter/map/reduce idiom) with ranges? `ranges::accumulate` is a possible candidate, but it is not "pipeable" and does not respect move-semantics (numerics-oriented).
- (5) `foldl_d` returns a `std::map`, which satisfies `ForwardRange`, but it will not compose with the downstream `group-by` because it's an rvalue. There's no `group_all_by` in ranges, so we'd have to dump the intermediate result into an lvalue first to apply a `sort`-action.
- (6,7) `transform, concat`: This is the same problem that we have already seen with "intersperse" example, where `range-v3` can't flatten a sequence of rvalue-containers.

Example 3: Transform-in-parallel

The function below is taken from aln_filter.cpp example. (which, by the way, showcases the usefulness of lazy data-stream manipulation in applicable use-cases).

The purpose of `lazy_transform_in_parallel` is to do the same job as plain `transform`, except each invocation of the transform-function is executed in parallel with up-to specified number of simultaneous async-tasks. (Unlike with c++17's parallelized `std::transform` we want this to work lazily with an `InputRange`.)

```
static auto lazy_transform_in_parallel = [](auto fn,
                                            size_t max_queue_size =
std::thread::hardware_concurrency())
{
    namespace fn = rangeless::fn;
    using fn::operators::operator%;
    assert(max_queue_size >= 1);
    return [max_queue_size, fn](auto inputs) // inputs can be an
lazy InputRange
    {
        return std::move(inputs)
```

```
        //-------------------------------------------------------------
---------
        // Lazily yield std::async invocations of fn.
      % fn::transform([fn](auto inp)
        {
            return std::async(std::launch::async,
                [inp = std::move(inp), fn]() mutable // mutable
because inp will be moved-from
                {
                    return fn(std::move(inp));
                });
        })
        //-------------------------------------------------------------
---------
        // Cap the incoming sequence of tasks with a seq of
_max_queue_size_-1
        // dummy future<...>'s, such that all real tasks make it
        // from the other end of the sliding-window in the next
stage.
      % fn::append(fn::seq([i = 1UL, max_queue_size]() mutable
        {
            using fn_out_t =
decltype(fn(std::move(*inputs.begin())));
            return i++ < max_queue_size ? std::future<fn_out_t>() :
fn::end_seq();
        }))
        //-------------------------------------------------------------
---------
        // Buffer executing async-tasks in a fixed-sized sliding
window;
        // yield the result from the oldest (front) std::future.
      % fn::sliding_window(max_queue_size)
      % fn::transform([](auto view) // sliding_window yields a view
into its queue
        {
            return view.begin()->get();
        });
    };
};
```

One would think that this has all the pieces to be implementable with ranges, but that is not the case. The obvious problem is that `view::sliding` requires a `ForwardRange`. Even if we decided to implement an "illegal" buffering version of `sliding`, there are more problems that are not visible in the code, but will manifest at runtime:

In `range-v3` the correct usage of `view::transform` is contingent on the following assumptions:

- It is cheap to recompute (This doesn't work for the first `transform` in the above example that takes and passes the input by-move and launches an async-task).
- It's OK to invoke it multiple times on the same input (This doesn't work for the second `transform`, where the call to `std::future::get` leaves it in invalid state, and so can only be called once).

If the transform-function is something like "add one" or "square an int" these assumptions are probably fine, but if the transform-function needs to query a database or spawn a process to run a heavy task, such assumptions are a little presumptuous.

This problem is what Jonathan described in the Terrible Problem Of Incrementing A Smart Iterator.

This behavior is not a bug, and is, apparently, by design – yet another reason why we can't have nice things with `range-v3`.

In `rangeless`, `fn::transform` neither calls the transform-function on the same input more than once, nor does it cache the result.

Note: `transform_in_parallel` is provided in the `rangeless` library. Compare implementation of a parallelized gzip compressor with rangeless (Ctrl+F pigz) vs. RaftLib.

What's the conclusion from all this?

Complexity of ranges.

We need a reasonably coherent language that can be used by "ordinary programmers" whose main concern is to ship great applications on time.
[Bjarne Stroustrup]

Ranges simplify the code for basic use cases, for example, you can write `action::sort(vec)` instead of `std::sort(vec.begin(), vec.end())`. However, beyond the most basic usages the complexity of the code increases exponentially.

For example, how would one implement the above-mentioned intersperse-adapter?

Let's look at the Haskell example first, just to have a reference point of what "simple" ought to look like.

```
intersperse ::  a -> [ a ] -> [ a ]
intersperse     _ [ ] = [   ]
intersperse     _ [ x ] = [ x ]
intersperse delim    (x:xs) = x : delim : intersperse delim xs
```

Even if you've never seen any Haskell in your life, you can probably figure out how that works.

Below are three different ways of doing it with `rangeless`. Just like the Haskell's signature `my_intersperse` takes a delim and returns a unary callable that can take some `Iterable` and return a sequence yielding the elements, interspersing `delim`.

A) As a generator-function:

```
auto my_intersperse = [](auto delim)
{
    return [delim = std::move(delim)](auto inputs)
    {
        return fn::seq([  delim,
                        inputs = std::move(inputs),
                            it = inputs.end(),
                       started = false,
                          flag = false]() mutable
        {
            if(!started) {
                started = true;
                it = inputs.begin();
            }
            return it == inputs.end() ? fn::end_seq()
                 :     (flag = !flag) ? std::move(*it++)
                 :                      delim;
        });
    };
};
```

B) By using `fn::adapt`, a facility in rangeless for implementing custom adapters

```
auto my_intersperse = [](auto delim)
{
    return fn::adapt([delim, flag = false](auto gen) mutable
    {
        return           !gen ? fn::end_seq()
             : (flag = !flag) ? gen()
             :                  delim;
```

```
    });
};
```

C) As composition of existing functions (what we attempted and failed to implement with range-views)

```
auto my_intersperse = [](auto delim)
{
    return [delim = std::move(delim)](auto inputs)
    {
        return std::move(inputs)
      % fn::transform([delim](auto inp)
        {
            return std::array<decltype(inp), 2>{{ std::move(inp),
delim }};
        })
      % fn::concat()
      % fn::drop_last(); // drop trailing delim
    };
};
```

D) We can also implement intersperse as a coroutine, without any help from `rangeless::fn`.

```
template<typename Xs, typename Delim>
static unique_generator<Delim> intersperse_gen(Xs xs, Delim delim)
{
    bool started = false;
    for (auto&& x : xs) {
        if(!started) {
            started = true;
        } else {
            co_yield delim;
        }
        co_yield std::move(x);
    }
};

auto my_intersperse = [](auto delim)
{
    return [delim](auto inps)
    {
        return intersperse_gen(std::move(inps), delim);
    };
};
```

All of the implementations are about the same in terms of code complexity. Now let's look at what the `range-v3` implementation looks like: intersperse.hpp. To me, personally, this

looks hypercomplex. If you're not sufficiently impressed, consider an implementation of a `cartesian-product` as a coroutine:

```
template<typename Xs, typename Ys>
auto cartesian_product_gen(Xs xs, Ys ys)
  -> unique_generator<std::pair<typename Xs::value_type,
                                typename Ys::value_type>>
{
    for(const auto& x : xs)
        for(const auto& y : ys)
            co_yield std::make_pair(x, y);
}
```

Compare the above with range-v3 implementation.

Writing views with range-v3 is supposed to be easy, but, as the examples show, the bar of what's considered "easy" in post-modern c++ has been raised to the heights not reachable by mere mortals.

The situation in the application code involving ranges is not any simpler.

Compare Haskell vs. Rust vs. rangeless vs. range-v3 implementations of a calendar-formatting app. Don't know about you, but the last implementation does not inspire me to ever have to understand or write code like this.

Note that in the `range-v3` example the authors break their own view copy-complexity requirements in `interleave_view` by having a `std::vector` field.

Range views leak abstraction

One of the big promises of ranges is abstracting-away the iterators. In our `rangeless` + coroutine implementations above we've successfully managed not having to deal with iterators directly in all cases except for (A) – manually capturing the input-range in the closure and then yielding its elements with `std::move(*it++)`

If you go back to the `range-v3` intersperse and calendar-app above and study in it more detail, you'll see that in implementation of views we end up dealing with iterators directly,

quite a lot in fact. Ranges don't save you from dealing with iterators directly beyond calling `sort` on a range or some such. On the contrary, it's "dealing with iterators, with extra steps".

Compile-time overhead

The `range-v3` library is infamous for its compile times. "On my machine" the compilation time for the above calendar example is over 20s, whereas the corresponding `rangeless` implementation compiles in 2.4s, 1.8s of which is just the `#include <gregorian.hpp>` – nearly an order of magnitude difference!

Compilation times are already an issue in every-day c++ development, and ranges don't just make it slightly worse! In my case this fact alone precludes any possibility of using ranges in production code.

The rangeless library

With `rangeless` I did not try to reinvent the wheel, and followed the design of streaming libraries in functional languages (Haskell's Data.List, Elixir's Stream, F#'s Seq, and LINQ).

Unlike in `range-v3`, there are no ranges, views, or actions – just passing of values from one function to the next through a chain of unary invokables, where a value is either a container or a sequence (input-range, bounded or unbounded).

There's a little bit of syntactic sugar:

```
operator % (Arg arg, Fn fn) -> decltype(fn(std::forward<Arg>(arg)))
auto x1 = std::move(arg) % f % g % h; // same as auto x1 =
h(g(f(std::move(arg))));
```

This is the equivalent of infix `operator &` in Haskell or `operator |>` in F#. This allows us to structure the code in a fashion congruent with the direction of the dataflow. It doesn't matter for a single-liner, but helps when the functions are multiline lambdas defined in-place.

Why `operator%` specifically, rather than `>>` or `|`, you wonder? The shopping-list of overloadable binary operators is not very long in C++, and the former tends to be heavily overloaded due to streams, and the pipe operator as well, usually for "smart"-flags, or "chaining" a.k.a point-free composition, as in ranges. I considered overloadable `operator->*`, but ultimately settled with `operator%` because given the context it's unlikely to be confused with integer-modulo, and also has `%=` counterpart that is useful to apply a state-change to LHS, e.g

```
vec %= fn::where(.../*satisfies-condition-lambda*/);
```

An input is either `seq` or a `Container`, and so is the output. E.g. `fn::sort` needs all elements to do its job, so it will dump entire input `seq` into a `std::vector`, sort it, and return as `std::vector`. A `fn::transform`, on the other hand, will wrap the input, taken by value, as `seq` that will lazily yield transformed input-elements. Conceptually this is similar to UNIX pipelines, with eager `sort` and lazy `sed`.

Unlike in `range-v3`, `input-ranges (sequences) are first-class citizens`. The issues of the concept-mismatches between arguments and parameters that we've seen in `range-v3` are non-existent (e.g. expecting `ForwardRange`, but received `InputRange`). Everything is composeable, as long as the value-types are compatible.

Epilog

I tried to use ranges to write expressive code. Am I the only one who ended up constantly "holding it wrong"?

I was quite surprised to learn that the committee accepted ranges into the c++20 standard and most c++ experts are excited about it. It's as if the issues of limited usability, code complexity, leaky abstractions and completely unreasonable compile-times are of no consequence whatsoever to the committee members?

I feel like there's a disconnect between the c++ experts that spearhead the development of the language and the common programmers that want simpler ways of doing complex

things. It seems to me that Bjarne Stroustrup's plea from Remember the Vasa! fell on deaf ears (again, my subjective opinion).

An Alternative Design to Iterators and Ranges, Using std::optional

Today's guest post is written by Vincent Zalzal. Vincent is a software developer working in the computer vision industry for the last 13 years. He appreciates all the levels of complexity involved in software development, from how to optimize memory cache accesses to devising algorithms and heuristics to solve complex applications, all the way to developing stable and user-friendly frameworks. You can find him online on Twitter or LinkedIn.

In a previous post, Jonathan presented what he calls the Terrible Problem Of Incrementing A Smart Iterator, or the TPOIASI. The problem occurs when an iterator that embeds logic in its `operator++` is composed with another iterator that performs some computation in its `operator*`. The TPOIASI is prevalent in code using the new C++ Ranges or ranges-v3.

I was intrigued by the problem, and decided to try solving it. While Jonathan decided to move the logic to smart output iterators to solve the problem, I decided to change the definition of range altogether.

Motivating example

Here is an example of the problem, using ranges-v3:

```
#include <iostream>
#include <vector>
#include <range/v3/all.hpp>

int times2(int n) {
    std::cout << "transform " << n << '\n';
    return n * 2;
}

bool isMultipleOf4(int n) {
    return n % 4 == 0;
}

int main() {
    std::vector<int> numbers = { 1, 2, 3, 4, 5 };
```

```
    std::vector<int> results; // output
    ranges::push_back(results,
          numbers | ranges::view::transform(times2)
                  | ranges::view::filter(isMultipleOf4));

    for (auto result : results)
        std::cout << result << ' ';
}
```

And here is the output:

transform 1

transform 2

transform 2 // transform called twice on 2

transform 3

transform 4

transform 4 // transform called twice on 4

transform 5

4 8

You can refer to Jonathan's article for a detailed explanation of what's going on. In summary, filter has to call both `operator++` and `operator*` of the underlying iterator in its own `operator++` to know when to stop, causing `transform` to apply its function (its `operator*`) twice per valid element: once in `filter`'s `operator++` and once in `filter`'s `operator*`.

```
auto FilterIterator::operator++() {
    do {
        ++curIt;
    } while (curIt != endIt && !pred(*curIt));
    return *this;
}
```

Suppose you are iterating over the filtered range like this:

```
for (auto it = filteredRange.begin(); it != filteredRange.end();
++it) {
    auto value = *it;
    // use value
}
```

The transform function is first called while performing `++it` to stop when the predicate is `true`, then it is called again right on the next line, in `*it`. Wouldn't it be nice if we could reuse the function evaluation in `++it` instead of calling `*it`?

Insight

Is it really necessary to have separate operations for advancing the iterator and evaluating its element?

If those two operations were to be merged into a single one, the spurious calls to the transform function would be avoided. Jonathan's solution using smart output iterators is actually doing all the work in the output iterator's `operator=`.

What if we could reinvent ranges from scratch without the need for low-level iterators? Could we leverage modern C++ features to iterate an input range with a single operation instead of two?

A Solution using std::optional

A solution is to represent an input range as a mutable view, i.e. a mutable structure that contains both the current position and the sentinel (the value returned by `std::end`). This way, we could define a single operation, let's call it `next`, that would return either the next element, or `std::nullopt` if the end of the range is reached.

```
// Non-owning input view based on STL iterators
template <typename InputIt, typename Sentinel>
struct InputRange {
    InputIt  current;
    Sentinel end;
    using value_type = typename
std::iterator_traits<InputIt>::value_type;

    std::optional<value_type> next() {
```

```
        if (current != end)
            return *current++;
        else
            return std::nullopt;
    }
};
```

I made the following design decisions to simplify the implementation:

- I only consider single-pass input ranges.
- `next()` return copies of the values, because optional references are not allowed... yet.

The downside of such a range is its size: it is twice the size of an STL iterator. This is only important if you are storing iterators in memory, though, which, in my opinion, is often not the best design anyway.

The filtered range is as easy to define as for standard ranges, maybe even easier, and it solves the problem presented in the motivating example.

```
// Range which filters elements of another range, based on a
predicate
template <typename Range, typename Pred>
struct FilteredRange {
    Range range;
    Pred  pred;
    using value_type = typename Range::value_type;

    std::optional<value_type> next() {
        while (const auto value = range.next())
        if (pred(*value))
            return value;
        return std::nullopt;
    }
};
```

Because `next` is performing both iteration and element evaluation, each element is evaluated exactly once.

The transformed range is even easier to define:

```
// Range which applies a transform to another range
template <typename Range, typename Func>
struct TransformedRange {
```

```
    Range range;
    Func  func;
    using value_type = decltype(func(*range.next()));

    std::optional<value_type> next() {
        if (const auto value = range.next())
            return func(*value);
        else
            return std::nullopt;
    }
};
```

With appropriate deduction guides, these structs are enough to implement the motivating example.

```
void withStructsOnly() {
    std::vector<int> numbers = { 1, 2, 3, 4, 5 };
    std::vector<int> results; // output
    auto filteredRange = FilteredRange{
                             TransformedRange{
                                 InputRange{numbers.begin(),
numbers.end()},
                                 times2
                             },
                             isMultipleOf4
                         };

    while (const auto value = filteredRange.next())
        results.push_back(*value);

    for (const auto value : results)
        std::cout << value << ' ';
}
```

No TPOIASI, success!

transform 1

transform 2

transform 3

transform 4

transform 5

48

Pipe syntax

I was happy with the result, but unsatisfied with the syntax. Under the encouragement of Jonathan, I implemented a basic mechanism to achieve a pipe syntax similar to the one in ranges-v3.

We would like to be able to replace this:

```
TransformedRange{SomeRange, times2}
```

by this:

```
SomeRange | transform(times2)
```

To achieve this, we must overload `operator|` to take any range as left-hand side operand, and an object returned by `transform` as right-hand side operand, that object temporarily holding the function to apply. Here is what it looks like, including the deduction guide for `TransformedRange`:

```
template <typename Range, typename Func>
TransformedRange(Range, Func) -> TransformedRange<Range, Func>;

template <typename Func>
struct TransformProxy {
    Func func;
};

template <typename Func>
auto transform(Func&& func) {
    return TransformProxy<Func>{std::forward<Func>(func)};
}

template <typename Range, typename Func>
auto operator|(Range&& range, TransformProxy<Func> proxy) {
    return TransformedRange{std::forward<Range>(range),
std::move(proxy.func)};
}
```

By doing the same thing for the `filter` function and adding a factory function to create the

input range, we get this much nicer-looking code:

```
auto filteredRange = make_range(numbers) | transform(times2) |
filter(isMultipleOf4);
```

Here is the full code listing. You can see it in action on Coliru.

```
#include <iterator>  // for iterator_traits, begin, end
#include <optional>
#include <utility>   // for forward, move

// Non-owning input view based on STL iterators
template <typename InputIt, typename Sentinel>
struct InputRange {
    InputIt  current;
    Sentinel end;

    using value_type = typename
std::iterator_traits<InputIt>::value_type;

    std::optional<value_type> next() {
        if (current != end)
            return *current++;
        else
            return std::nullopt;
    }
};

template <typename InputIt, typename Sentinel>
InputRange(InputIt, Sentinel) -> InputRange<InputIt, Sentinel>;

// Factory function taking anything with begin/end support and
returning a mutable view
template <typename T>
auto make_range(T&& c) {
    return InputRange{std::begin(c), std::end(c)};
}

// Range which filters elements of another range, based on a
predicate
template <typename Range, typename Pred>
struct FilteredRange {
    Range range;
    Pred  pred;

    using value_type = typename Range::value_type;

    std::optional<value_type> next() {
        while (const auto value = range.next())
            if (pred(*value))
                return value;
        return std::nullopt;
```

```
    }
};

template <typename Range, typename Pred>
FilteredRange(Range, Pred) -> FilteredRange<Range, Pred>;

// Range which applies a transform to another range
template <typename Range, typename Func>
struct TransformedRange {
    Range range;
    Func  func;

    using value_type = decltype(func(*range.next()));

    std::optional<value_type> next() {
        if (const auto value = range.next())
            return func(*value);
        else
            return std::nullopt;
    }
};

template <typename Range, typename Func>
TransformedRange(Range, Func) -> TransformedRange<Range, Func>;

// Pipe-syntax enabler structs and operator overloads
template <typename Func>
struct TransformProxy {
    Func func;
};

template <typename Func>
auto transform(Func&& func) {
    return TransformProxy<Func>{std::forward<Func>(func)};
}

template <typename Range, typename Func>
auto operator|(Range&& range, TransformProxy<Func> proxy) {
    return TransformedRange{std::forward<Range>(range),
std::move(proxy.func)};
}

template <typename Pred>
struct FilterProxy {
    Pred pred;
};

template <typename Pred>
auto filter(Pred&& pred) {
    return FilterProxy<Pred>{std::forward<Pred>(pred)};
}

template <typename Range, typename Pred>
auto operator|(Range&& range, FilterProxy<Pred> proxy) {
```

```
    return FilteredRange{std::forward<Range>(range),
std::move(proxy.pred)};
}

// Motivating example
#include <iostream>
#include <vector>

int times2(int n) {
    std::cout << "transform " << n << '\n';
    return n * 2;
}

bool isMultipleOf4(int n) {
    return n % 4 == 0;
}

void withStructsOnly() {
    std::vector<int> numbers = { 1, 2, 3, 4, 5 };
    std::vector<int> results; // output

    auto filteredRange = FilteredRange{
        TransformedRange{
            InputRange{numbers.begin(), numbers.end()},
            times2
        },
        isMultipleOf4
    };

    while (const auto value = filteredRange.next())
        results.push_back(*value);

    for (const auto value : results)
        std::cout << value << ' ';
}

void withPipeSyntax() {
    std::vector<int> numbers = { 1, 2, 3, 4, 5 };
    std::vector<int> results; // output

    auto filteredRange = make_range(numbers) | transform(times2) |
filter(isMultipleOf4);

    while (const auto value = filteredRange.next())
        results.push_back(*value);

    for (const auto value : results)
        std::cout << value << ' ';
}

int main() {
    std::cout << "With structs only:\n";
    withStructsOnly();
    std::cout << "\nWith pipe syntax:\n";
```

```
    withPipeSyntax();
}
```

STL algorithms and range-based for loop

You might wonder why I am not using `std::copy` to push back elements into the output vector, or why I create a temporary variable to hold the range. This is because `InputRange`, `FilteredRange` and `TransformedRange` do not play nicely with existing C++ features and libraries. The range-based for statement:

```
for (for-range-declaration : for-range-initializer) statement
```

is currently equivalent to:

```
{
    auto &&__range = for-range-initializer ;
    auto __begin = begin-expr ;
    auto __end = end-expr ;
    for ( ; __begin != __end; ++__begin ) {
        for-range-declaration = *__begin;
        statement
    }
}
```

Let's imagine an alternate universe where the range-based for loop would instead be based on `next`:

```
{
    auto &&__range = for-range-initializer ;
    while (auto __value = std::next(__range)) { // same as
__range.next()
        for-range-declaration = *__value;
        statement
    }
}
```

In this C++ fantasy land, STL algorithms would also have overloads taking such a range as first argument. Then, we would finally get this coveted version of the motivating example:

```
// Fantasy, this does not compile.
int main() {
    std::vector<int> numbers = { 1, 2, 3, 4, 5 };
```

```
    std::vector<int> results; // output
    std::copy(make_range(numbers) | transform(times2) |
filter(isMultipleOf4),
              std::back_inserter(results));
    for (const auto value : results)
        std::cout << value << ' ';
    // Or, without even using a temporary output vector:
    for (const auto value : make_range(numbers)
             | transform(times2)
             | filter(isMultipleOf4))
        std::cout << value << ' ';
}
```

Performance

You wouldn't be a *real* C++ programmer if you weren't caring about performance, would you? You will be happy to know that most recent compilers see through all the abstraction layers of proxy objects and `std::optional`s. gcc-trunk in particular generates almost the exact same code as a handwritten loop doing all computations inline, as can be seen on Compiler Explorer. Very impressive!

Note that, at the time of writing, gcc-trunk seems to be the only version of x86-64 gcc on Compiler Explorer to generate that code, so your mileage may vary.

Conclusion

In the book *From Mathematics to Generic Programming*, Alexander Stepanov and Daniel Rose describe the *Law of Useful Return*:

If you've already done the work to get some useful result, don't throw it away. Return it to the caller. This may allow the caller to get some extra work done "for free".

For example, since C++11, `std::rotate` returns an iterator to the new position of the previously-last iterator. Maybe it won't be used, but it was already computed anyway.

In this article, I applied this programming principle to `operator++` for filter iterators. When incrementing the iterator, its current value must be evaluated to determine if it satisfies the predicate or not. That evaluated value should be returned instead of being discarded.

By combining both `operator++` and `operator*` into a single function, it is possible to both increment the iterator *and* return the evaluated value, thus avoiding the Terrible Problem Of Incrementing A Smart Iterator: evaluating the value twice. Moreover, I think any programmer that once implemented an iterator class will agree that it is not a trivial task, and implementing `FilteredRange` and `TransformedRange` above required quite less boilerplate code.

Thinking out of the box when solving toy problems may sometimes lead to interesting insights. I hope you had as much fun reading this article as I had fun writing it. Thanks to Tim van Deurzen for providing constructive feedback, and thanks to Jonathan for giving me the opportunity again of writing a guest post on his blog. Happy coding!