SMART OUTPUT ITERATORS

- **How Smart Output Iterators Avoid the TPOIASI**
- **Unzipping a Collection of Tuples with the “unzip” Smart Output Iterator**
- **Applying Several Transforms in One Pass on a Collection**
- **Is Unzip a Special Case of Transform?**
- **Partitioning Data with Output Iterators in C++**

Fluent {C++}

How Smart Output Iterators Avoid the TPOIASI

In a previous article we saw the TPOIASI, or Terrible Problem Of Incrementing A Smart Iterator, that could incur a performance cost in code that uses range adaptors. Today, we'll see how smart output iterators fare with the TPOIASI (spoiler: they have a way to avoid the problem).

Now if you're wondering what smart iterators, smart output iterators or the Terrible Problem Of Incrementing Them is, here is a little refresher.

The TPOIASI

The TPOIASI occurs when an iterator that embeds logic in its `operator++` (for instance, advancing to the next element that satisfies a predicate), is plugged onto another iterator, for example one that applies a function in its `operator*`.

In a range-style code, the situation looks like this:

```
// 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)
{
```

```
    std::cout << "transform " << n << '\n';
    return n * 2;
}

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

(note the trace in `times2`).

The code outputs:

```
transform 1
transform 2
transform 2
transform 3
transform 4
transform 4
transform 5
4 8
```

For some elements, `2` and `4`, the function is called more than once. This is a problem. And a terrible one because it is – in my opinion – structural to this range adaptor.

We had seen that the source of the problem is that the `operator++` of `filter` that has to peek ahead to know where to stop, and then its `operator*` calls up the `transform` function again.

If you'd like to read more about the Terrible Problem Of Incrementing A Smart Iterator, you can have a look at its dedicated article.

Smart output iterators

Smart output iterators are a symmetrical approach to range adaptors, to manipulate collections in C++. This means that while range adaptors operate on **input iterators** and can funnel data into an STL algorithm, smart output iterators put some logic inside of the **output iterators** of an algorithm.

Take `std::back_inserter` for instance. It is an output iterator that embeds a `push_back` to a container. Smart output iterators generalise this idea by allowing output iterators to apply

functions, filter on predicates, and a lot of other fancy treatments, to the data coming out of STL algorithms.

For instance, the equivalent code to the one above that used range adaptors would be, with smart output iterators:

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

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

//Apply transform and filter
auto oIsMultiple4 = make_output_filter(isMultiple4);
auto oTimes2 = make_output_transformer(times2);

copy(numbers, oTimes2(oIsMultiple4(back_inserter(results))));

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

Now do smart output iterators suffer from the TPOIASI? Do they call the function in `transform` multiple times?

When we look at the implementation of the output iterator that filters, its `operator++` and `operator*` implementations are pretty ascetic (like for all output iterators):

```
template<typename Iterator, typename Predicate>
class output_filter_iterator
{
public:
    explicit output_filter_iterator(Iterator iterator, Predicate predicate)
: iterator_(iterator), predicate_(predicate) {}

    output_filter_iterator& operator++(){ ++iterator_; return *this; }
    output_filter_iterator& operator*(){ return *this; }

    template<typename T>
    output_filter_iterator& operator=(T const& value)
    {
        if (predicate_(value))
        {
            *iterator_ = value;
        }
        return *this;
```

```
    }
private:
    Iterator iterator_;
    Predicate predicate_;
};
```

No checking of the predicate, no reading from the underlying iterator.

Will this be enough to make them immune to the Terrible Problem?

Let's run that code to find out.

Smart output iterators and the TPOIASI

Running the code with the same trace:

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

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

gives this output:

```
transform 1
transform 2
transform 3
transform 4
transform 5
4 8
```

No multiple calls to the function!

Does that mean that smart output iterators are immune to the Terrible Problem?

It's not that simple. The above case appends data to an empty `vector`, with the help of a `back_inserter`. But if we change the use case a little, by **writing into the vector** rather than appending to it:

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

// Output vector
std::vector<int> results = {0, 0, 0, 0, 0};

//Apply transform and filter
auto oIsMultiple4 = make_output_filter(isMultiple4);
auto oTimes2 = make_output_transformer(times2);

copy(numbers, oTimes2(oIsMultiple4(begin(results))));

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

We would expect this:

```
4 8 0 0 0
```

But the result we get is in fact that:

```
0 4 0 8 0
```

This is a bug. It comes from the `operator++` that increments the underlying iterator even if the smart output iterator ends up not writing to it (in the case where the value it is passed doesn't satisfy the predicate).

Let's attempt to fix this by changing the implementation of `operator++` from this:

```
output_filter_iterator& operator++(){ ++iterator_; return *this; }
```

as it was above, to that:

```
output_filter_iterator& operator++(){ return *this; }
```

By **not** incrementing the underlying iterator.

The the result we get is now this:

```
8 0 0 0 0
```

This is still not good, because we're *never* incrementing the underlying iterator, therefore we're constantly writing at the same position.

No, we'd need to increment the filter iterator **only if is has sent something to its underlying iterator**. Let's just write it then:

```
template<typename Iterator, typename Predicate>
class output_filter_iterator
{
public:
    explicit output_filter_iterator(Iterator iterator, Predicate predicate)
: iterator_(iterator), predicate_(predicate) {}

    output_filter_iterator& operator++(){ return *this; }
    output_filter_iterator& operator*(){ return *this; }

    template<typename T>
    output_filter_iterator& operator=(T const& value)
    {
        if (predicate_(value))
        {
            *iterator_ = value;
            ++iterator_;
        }
        return *this;
    }
private:
    Iterator iterator_;
    Predicate predicate_;
};
```

Now when we run the code we get:

```
4 8 0 0 0
```

And does the case with `back_inserter` still work? Let's run it:

```
4 8
```

It does still work.

It all looks good except there is a nagging question left:

Is this OK?

Implementing the `operator++` by incrementing the underlying sounded natural. Indeed, imagine that an algorithm decided to increment the output iterator twice before assigning it. A `std::vector` iterator would skip an element, but our smart output iterator would completely be oblivious to that double-increment.

It turns out that it's ok, because algorithms are not allowed to increment an output iterator twice without calling `operator=` in between. Indeed, as we can read on cppreference.com, "Assignment through an output iterator is expected to alternate with incrementing. Double-increment is undefined behavior".

I may well miss something, but this makes this implementation look ok to me, and **smart output iterators have avoided the TPOIASI**, which looks like a good sign for their design.

If you'd like to see the code of the smart output iterators library, it's up on GitHub.

Unzipping a Collection of Tuples with the “unzip” Smart Output Iterator

Smart output iterators are output iterators that do more than just sending a piece of data from an STL algorithm to a container. They can embed logic that relieves the algorithm of some of its responsibilities.

We have already seen examples of smart output iterators that apply a function or filter on a predicate.

Now let’s see an example of smart output iterator that breaks down pairs and tuples, so that all the first elements go to one direction, all the second elements to another direction, and so on.

Two motivating cases: separating key from values, and transposing a collection a tuples

Let’s see two motivating examples for breaking down collections of pairs and tuples into specific containers.

Pairs

A `std::map` is a sorted collection of `std::pair`s, whose `first`s are keys and `second`s are values. We want to send the keys and the values of the map to two distinct containers. And to leverage on the power of smart output iterators, let’s say that we also want to apply a function only on values.

To illustrate, let’s create a map that associates strings to numbers:

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

We would like to:

- send the keys to `keys`,
- send the values **in upper case** to `values`

with `keys` and `values` starting as empty containers:

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

For this we need to implement the `unzip` output iterator. We will also use the `transform` iterator (formerly called `output_transformer`) to apply a function to the output of the `unzip` iterator:

```
auto const toUpper = fluent::output::transform(toUpperString);

std::copy(begin(entries), end(entries),
          unzip(back_inserter(keys),
                toUpper(back_inserter(values))));
```

`toUpperString` is a function that takes a `std::string` and returns a `std::string` that is the former one in upper case. It can be implemented like this:

```
std::string toUpperString(std::string const& s)
{
    std::string upperString;
    std::transform(begin(s), end(s), std::back_inserter(upperString),
[](char c){ return std::toupper(c); });
    return upperString;
}
```

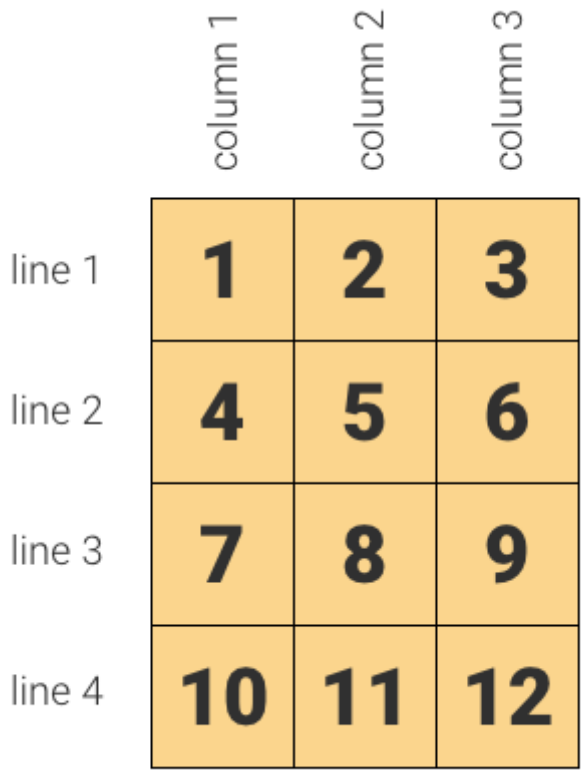
And we would like `keys` to contain `{1, 2, 3, 4, 5}`, and values to contain `{"ONE", "TWO", "THREE", "FOUR", "FIVE"}`.

Tuples

A more generic use case would use tuples instead of pairs. Here is a collection of tuples:

```
std::vector<std::tuple<int, int, int>> lines = { {1, 2, 3}, {4, 5, 6}, {7,
8, 9}, {10, 11, 12} };
```

In our example, this collection represents the lines of a table: the first line is 1 2 3, the second line is 4 5 6, and so on.



Let's extract the columns of the table. To do this, we need to extract the first elements of each line and put them into a `column1` container, then the second elements of each line and put them into a `column2` container, and so on.

So our target code will be:

```
std::vector<int> column1, column2, column3;

std::copy(begin(lines), end(lines),
          unzip(back_inserter(column1),
                back_inserter(column2),
                back_inserter(column3)));
```

And we expect `column1` to hold `{1, 4, 7, 10}`, `column2` to hold `{2, 5, 8, 11}`, and `column3` to hold `{3, 6, 9, 12}`.

Now that we have those two use cases for it, let's implement the `unzip` output iterator.

The unzip output iterator

`unzip` will follow the typical implementation of smart output iterators:

- the constructor keeps track of the underlying iterators to send data to,
- `operator*` returns the object itself, so that…
- …`operator=` is called by the user (e.g. STL algorithm) and can perform the action of sending data off to the underlying iterators,
- `operator++` forwards the increment to the underlying iterators.

So let's start with the constructor:

```
template<typename... Iterators>
class output_unzip_iterator
{
public:
    explicit output_unzip_iterator(Iterators... iterators) :
iterators_(std::make_tuple(iterators...)) {}

private:
    std::tuple<Iterators...> iterators_;
};
```

We keep all the underlying iterators into a `tuple`. Indeed, there could be any number of underlying iterators.

The `operator*` does its job of allowing our smart output iterator to stay in the game when dereferenced:

```
output_unzip_iterator& operator*(){ return *this; }
```

The action then happens in `operator=`, when the STL algorithms assigns to what is returned by dereferencing the iterator (so here, the iterator itself). Let's start with the simpler case of sending an `std::pair` to our iterator:

```
output_unzip_iterator& operator=(std::pair<First, Second> const& values)
{
    *std::get<0>(iterators_) = values.first;
    *std::get<1>(iterators_) = values.second;
    return *this;
}
```

We forward the first (resp. second) of the incoming pair to the first (resp. second) underlying iterator.

The overload of `operator=` that receives a `std::tuple` is less straightforward to implement. Its prototype looks like this:

```
template<typename... Ts>
output_unzip_iterator& operator=(std::tuple<Ts...> const& values)
{
```

And in this function, we need to send each element of the incoming **tuple** to its corresponding element in our **tuple** of underlying iterators.

One way of formulating this is to apply to each pair of respective elements of those tuples a function that takes a value and an iterator, and that sends that value to that iterator.

So the problem comes down to applying a function taking two parameters to respective elements coming from two tuples.

Applying a function to the elements of two tuples

Note: We're going to delve into template metaprogramming and variadic templates here. I'm not an expert, and if you know how to improve what follows, I'm happy to hear your feedback!

To apply a function to the elements of **one** tuple, C++17 offers `std::apply`. But before C++17, there was a way to emulate `std::apply`. We are going to look into this implementation, and adapt it for elements coming from **two** tuples.

To apply a function to the elements of a tuple, we can 1) unwrap the tuple into a variadic pack and 2) pass the contents of the variadic pack as arguments to a function.

Unwrapping the tuple into a variadic pack

To do this, we use C++14 `index_sequence`:

```
template <class F, class Tuple1, class Tuple2>
constexpr decltype(auto) apply2(F&& f, Tuple1&& t1, Tuple2&& t2)
```

```
{
    return apply2_impl(std::forward<F>(f), std::forward<Tuple1>(t1),
std::forward<Tuple2>(t2),

std::make_index_sequence<std::tuple_size<std::remove_reference_t<Tuple1>>::
value>{});
}
```

Passing the contents of a variadic pack as arguments to a function

`apply2_impl` is a function that unwraps the contents of the tuples and pass them as parameters to `f`:

```
template <class F, class Tuple1, class Tuple2, std::size_t... I>
F apply2_impl(F&& f, Tuple1&& t1, Tuple2&& t2, std::index_sequence<I...>)
{
    return
(void)std::initializer_list<int>{(std::forward<F>(f)(std::get<I>(std::forwa
rd<Tuple1>(t1)), std::get<I>(std::forward<Tuple2>(t2))),0)...}, f;
}
```

I reckon it is Sean Parent who came up with the technique of passing the contents of a variadic pack as arguments to a function without C++17. The above adapts that technique to a function that takes two parameters.

If you're not familiar with variadic templates, I realize the above code must look not very different from this:

And it's OK. You don't need to understand those details to get the general meaning of the `unzip` iterator, and to use it. However, this manipulation of compile time collections is an interesting topic, and we'll dive into it in a later post with more explanations.

Anyway, the body of `operator=` for our `unzip` iterator is now:

```
output_unzip_iterator& operator=(std::tuple<Ts...> const& values)
{
    apply2([](auto&& value, auto&& iterator){ *iterator = value; }, values,
iterators_);
    return *this;
}
```

One last thing to implement is the increment operator: `operator++`. Here we make it forward the increment to its underlying iterators. So we need to apply a function that calls ++ on each element of the tuple. We could use `std::apply` in C++17, and in C++14 we can resort to an implementation with the technique we saw before:

```
template <class F, class Tuple, std::size_t... I>
F apply_impl(F&& f, Tuple&& t, std::index_sequence<I...>)
{
    return
(void)std::initializer_list<int>{(std::forward<F>(f)(std::get<I>(std::forwa
rd<Tuple>(t))),0)...}, f;
}

template <class F, class Tuple>
constexpr decltype(auto) apply(F&& f, Tuple&& t)
{
    return apply_impl(std::forward<F>(f), std::forward<Tuple>(t),

std::make_index_sequence<std::tuple_size<std::remove_reference_t<Tuple>>::v
alue>{});
}
```

And we use it this way:

```
output_unzip_iterator& operator++()
{
    detail::apply([](auto&& iterator){ ++iterator; }, iterators_);
    return *this;
}

output_unzip_iterator& operator++(int){ ++*this; return *this; }
```

Finally let's not forget the aliases for iterators:

```
using iterator_category = std::output_iterator_tag;
using value_type = void;
using difference_type = void;
using pointer = void;
using reference = void;
```

And the actual `unzip` function that instantiates the iterator:

```
template<typename... Iterators>
output_unzip_iterator<Iterators...> unzip(Iterators... iterators)
{
    return output_unzip_iterator<Iterators...>(iterators...);
}
```

And we're good to go.

Unzipping pairs and tuples

Let's now test our new iterator!

Our first use case was breaking down a collection of pairs into a collection of keys and a collection of values, and apply a function on values:

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

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

auto const toUpper = fluent::output::transform(toUpperString);

std::copy(begin(entries), end(entries),
          unzip(back_inserter(keys),
                toUpper(back_inserter(values))));
```

When we output the contents of `keys` we now get:

```
1 2 3 4 5
```

And when we output the contents of `values` we get:

```
ONE TWO THREE FOUR FIVE
```

And our second case was using tuples, to breaks a collection of lines into a collection of columns:

```
std::vector<std::tuple<int, int, int>> lines = { {1, 2, 3}, {4, 5, 6}, {7,
8, 9}, {10, 11, 12} };
std::vector<int> column1, column2, column3;

std::copy(begin(lines), end(lines),
          unzip(back_inserter(column1),
                back_inserter(column2),
                back_inserter(column3)));
```

When we output the contents of `column1` we get:

```
1 4 7 10
```

The outputs of `column2` give:

```
2 5 8 11
```

And those of `column3` are:

```
3 6 9 12
```

If you want to have a closer look at the code, you can check out the smart output iterators library, the implementation of the `unzip` iterator, and the tests associated to it.

Applying Several Transforms in One Pass on a Collection

Applying a function to each element of a collection and outputting the results into another collection is a very common thing to do, in C++ or elsewhere.

In C++, we have the `std::transform` algorithm to do this, a central piece of the STL algorithms library.

To illustrate, consider the following program:

```
#include <algorithm>
#include <iterator>
#include <vector>
#include <iostream>

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

int main()
{
    auto const inputs = std::vector<int>{0, 1, 2, 3, 4, 5};
    auto outputs = std::vector<int>{};

    std::transform(begin(inputs), end(inputs), back_inserter(outputs),
times2);

    for (auto const& output : outputs)
    {
        std::cout << output << ' ';
    }
}
```

It outputs this:

```
0 2 4 6 8 10
```

The output iterator we're using here, `std::back_inserter`, forwards the data it receives to the `push_back` method of the `outputs` collection.

But can we apply **several functions** to each elements of the collection, and output the results into several collections?

With standard algorithms, we can't. But with smart output iterators, we could.

Smart output iterators, you said?

When we explored smart output iterators, we saw that we could write the above code differently, by pushing the logic out of the algorithm and towards the output iterator.

The code using smart output iterators and equivalent to the previous example would be this:

```
std::vector<int> input = {1, 2, 3, 4, 5};
std::vector<int> results;

auto const times2 = fluent::output::transform([](int i) { return i*2; });
std::copy(begin(input), end(input), times2(back_inserter(results)));
```

Note that we no longer use `std::transform` but rather `std::copy` which does less things, and the logic has been transferred to `times2`, which is now an outputs iterator. `times2` receives data from `std::copy`, multiplies it by 2, and sends the result on to the good old `back_inserter`.

This is no longer standard C++. This relies on the Smart Output Iterators library, that provides amongst other things the `transform` iterator. For more details about smart outputs iterators, you can check out the library, or this introductory blog post.

The characteristic aspect of smart outputs iterators are their position: in the **output** of the algorithm. Let's take advantage of their position to do something that an algorithm can't do: applying several functions on the same collection.

Applying several functions to the elements of a collection

This is something that happens in our everyday programming life: you have several functions, and you'd like to apply each of them to the elements of your collection.

Let's enrich the `transform` output iterator so that it supports **more than one function**. For example, we'd like to be able to write code like this:

```
std::vector<int> input = {0, 1, 2, 3, 4, 5};

auto const times234 = fluent::output::transform([](int i) { return i*2; },
                                                [](int i) { return i*3; },
                                                [](int i) { return i*4; });

std::vector<int> results1;
std::vector<int> results2;
std::vector<int> results3;

std::copy(begin(input), end(input),
          times234(back_inserter(results1),
                   back_inserter(results2),
                   back_inserter(results3)));
```

This would apply each of the 3 functions defined in the output iterators to each of the elements of the collections, and dispatch the results in 3 corresponding collections (`results1`, `results2`, `results3`).

So if we print out the contents of the output collections, for example with this code:

```
for (auto const& result : results1) { std::cout << result << ' '; }
std::cout << '\n';
for (auto const& result : results2) { std::cout << result << ' '; }
std::cout << '\n';
for (auto const& result : results3) { std::cout << result << ' '; }
std::cout << '\n';
```

We'd like it to display this output:

```
0 2 4 6 8 10
0 3 6 9 12 15
0 4 8 12 16 20
```

Can we do this? Yes we can, and we'll see the implementation in just a moment.

But before that, let's reflect on the interest of this feature. Let's compare the code using standard algorithms to achieve the same thing:

```
std::vector<int> input = {0, 1, 2, 3, 4, 5};

std::vector<int> results1;
std::vector<int> results2;
std::vector<int> results3;

std::transform(begin(input), end(input), back_inserter(results1), [](int i)
{ return i*2; });
std::transform(begin(input), end(input), back_inserter(results2), [](int i)
{ return i*3; });
std::transform(begin(input), end(input), back_inserter(results3), [](int i)
{ return i*4; });
```

This code can be seen as more straightforward that the one above using smart output iterators because it just repeats the same pattern. And it can also be seen as less straightforward because it makes several passes on the same collection, whereas the one using smart output iterators only makes one pass.

The interest of using smart output iterators becomes even clearer when there is more than just applying a function. If you'd like to use filters, for example (or any other output iterator in the library, including applying other functions with the `transform` iterator), the code using smart output iterators would look like this:

```
std::copy(begin(input), end(input),
          times234(aFilter(back_inserter(results1)),
                   back_inserter(results2),
                   anotherFilter(back_inserter(results3))));
```

Whereas using the standard algorithms doesn't scale well:

```
std::transform(begin(input), end(input),
back_inserter(notFilteredResults1), [](int i) { return i*2; });
std::copy_if(begin(notFilteredResults1), end(notFilteredResults1),
back_inserter(results1), aFilter);
std::transform(begin(input), end(input), back_inserter(results2), [](int i)
{ return i*3; });
std::transform(begin(input), end(input),
back_inserter(notFilteredResults3), [](int i) { return i*4; });
std::copy_if(begin(notFilteredResults3), end(notFilteredResults3),
back_inserter(results3), anotherFilter);
```

Let's now implement the possibility for the `transform` output iterator to have multiple outputs.

Implementing the multiple transform output iterator

We'll pick up where we left off in the introductory blog post: we have a transform output iterator that already support one output:

```
template<typename Iterator, typename TransformFunction>
class output_transform_iterator
{
public:
    using iterator_category = std::output_iterator_tag;
    explicit output_transform_iterator(Iterator iterator, TransformFunction
transformFunction) : iterator_(iterator),
transformFunction_(transformFunction) {}
    output_transform_iterator& operator++(){ ++iterator_; return *this; }
    output_transform_iterator& operator++(int){ ++*this; return *this; }
    output_transform_iterator& operator*(){ return *this; }
    template<typename T>
    output_transform_iterator& operator=(T const& value)
    {
        *iterator_ = transformFunction_(value);
        return *this;
    }
private:
    Iterator iterator_;
    TransformFunction transformFunction_;
};
```

The iterator contains two things:

- another iterator, to which it sends its results (for example it can be a `back_inserter`),
- the function to apply (which can also be a lambda–it is defined as a template parameter).

To have several outputs, the iterator must now contains:

- a *collection* of iterators to send results to,
- a *collection* of functions to apply.

And we must fit all this in the template parameter. The template parameters for one output look like this:

```
template<typename Iterator, typename TransformFunction>
```

It would be nice to be able to write then:

```
template<typename... Iterators, typename... TransformFunctions>
```

But we can't: C++ requires that the template parameters variadic pack be at the end of the template parameters (and as a result, there can be only one variadic pack).

To work around this constraint, we can pack up one group of parameters into one parameter, by using a tuple. Let's make this appear in its name:

```
template<typename TransformFunctionTuple, typename... Iterators>
```

We choose to pack up the functions, because it will make the implementation of other parts of the iterator easier.

As a result, the data members of the iterator, that used to be those:

```
Iterator iterator_;
    TransformFunction transformFunction_;
```

Now become these:

```
std::tuple<Iterators...> iterators_;
    TransformFunctionTuple transformFunctionTuple_;
```

And we expect `TransformFunctionTuple` to be a `std::tuple` of functions and/or lambdas to apply.

We now have to apply each function to the value incoming in `operator=`, and send the result to the corresponding output iterator.

For this we need to be able to apply a function to the elements of two tuples. We already came across this need in the past, when implementing the `unzip` output iterator. We came up then with the `apply2` function. You can check out the details of its implementation there.

By using `apply2`, the implementation of `operator=` goes from this:

```
*iterator_ = transformFunction_(value);
```

To this:

```
apply2([&value](auto&& function, auto&& iterator){ *iterator =
function(value); },
           transformFunctionTuple_,
           iterators_);
```

The rest of the adaptation consists in passing on the variadic template parameters from the `transform` function that creates the output iterator to the actual iterator class above `output_transform_iterator`. They don't contain any specific difficulty and you can see them in the commit introducing the feature in the library.

A new range of possibilities

This feature of outputting the results of several functions to several outputs seems like an important addition to the smart output iterators library.

For example, the `unzip` output iterator, that takes a tuple (or a pair) and sends its various pieces to as many output collections sounds like an application of our new transform iterator. The functions to apply would be the `std::get<N>` functions (or `.first` and `.second` for the `std::pair`).

To explore this in more detail, in the next article we will implement the `unzip` output iterator with the `transform` output iterator.

Is Unzip a Special Case of Transform?

In the Smart Output Iterators library, the `unzip` output iterator allows to send the various elements contained in tuples or pairs to as many output collections:

```
std::vector<std::tuple<int, int, int>> lines = { {1, 2, 3}, {4, 5, 6}, {7,
8, 9}, {10, 11, 12} };
std::vector<int> column1, column2, column3;

std::copy(begin(lines), end(lines),
fluent::output::unzip(back_inserter(column1), back_inserter(column2),
back_inserter(column3)));
```

This is a way to transpose a collection of lines into a collection of columns. Indeed, after executing the above code, `column1` contains `{1, 4, 7, 10}`, `column2` contains `{2, 5, 8, 11}`, and `column3` contains `{3, 6, 9, 12}`.

`unzip` also applies to maps, because they contains `std::pair`s of keys and values:

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

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

std::copy(begin(entries), end(entries),
fluent::output::unzip(back_inserter(keys), back_inserter(values)));
```

After executing this code, `keys` contains `{1, 2, 3, 4, 5}`, and `values` contains `{"one", "two", "three", "four", "five"}`.

For more about the unzip iterator, check out its dedicated post.

The transform iterator with multiple outputs

The smart output iterators library also has a `transform` output iterator. Its job is to apply a function to the data it receives, and to send the result on to another iterator:

```
std::vector<int> input = {1, 2, 3, 4, 5};
std::vector<int> results;

auto const times2 = fluent::output::transform([](int i) { return i*2; });

std::copy(begin(input), end(input), times2(back_inserter(results)));
```

After this code, `results` contains `{2, 4, 6, 8, 10}`.

For more about the `transform` iterator and about smart output iterators in general, check out this introductory post.

More recently, we generalised the `transform` output iterator so that it can take several functions to apply to each element of the collection, and send their results to as many output iterators:

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

auto const multiply = fluent::output::transform([](int i) { return i*2; },
                                                [](int i) { return i*3; },
                                                [](int i) { return i*4; });

std::vector<int> results1;
std::vector<int> results2;
std::vector<int> results3;

std::copy(begin(input), end(input), multiply(std::back_inserter(results1),
std::back_inserter(results2), std::back_inserter(results3)));
```

After executing this code, `expected1` contains `{2, 4, 6, 8, 10}`, `expected2` contains `{3, 6, 9, 12, 15}`, and `expected3` contains `{4, 8, 12, 16, 20}`.

Given all this, don't you think that `unzip` seems like a special case of `transform`?

Indeed, `unzip` consists in applying `std::get<0>` on the incoming tuple or pair and sending the result to one output iterator, applying `std::get<1>` and sending its results to another output, applying `std::get<2>` and sending its result to yet another output, and so on.

It sounds as if we could implement `unzip` with `transform`, `std::get` and a pinch of variadic templates. Let's try to code this up.

Implementing unzip with transform

If you look back at the first example of `unzip` above, you can see it is used this way:

```
unzip(back_inserter(column1), back_inserter(column2),
back_inserter(column3))
```

The prototype of `unzip` is this:

```
template<typename... Iterators>
auto unzip(Iterators... iterators)
{
    //...
```

We need to keep this prototype, and implement the function with the `transform` output iterator.

To do this we need to do two things:

- create the `transform` output iterator containing the functions to apply (the `std::get<I>`s)
- apply it to the `iterators...` pack

The second one being the easiest, let's focus on the first one: creating the `transform` output iterator.

As a reminder, the `transform` output iterator takes its functions this way:

```
transform([](int i) { return i*2; },
          [](int i) { return i*3; },
          [](int i) { return i*4; });
```

A variadic pack of integers

It would be nice to write something like `transform(std::get<Is>...)`, but for this we need a variadic pack of `Is...` going from `0` to the number of elements in the `Iterators...` pack minus one.

The C++ standard component that creates variadic packs of consecutive integers is `make_index_sequence`. Let's use it to create the pack of integers by passing it `sizeof...(Iterators)`, which is the number of elements in the `Iterators...` pack:

```
template<size_t... Is>
auto make_transform(std::index_sequence<Is...> const&)
{
    // see below
}

template<typename... Iterators>
auto unzip(Iterators... iterators)
{
    return
make_transform(std::make_index_sequence<sizeof...(Iterators)>{})(iterators.
..);
}
```

A better options, as suggested by Darell (who goes by the the Twitter handle of @beached_whale), is to use the more direct `std::index_sequence_for`:

```
template<typename... Iterators>
auto unzip(Iterators... iterators)
{
    return
make_transform(std::index_sequence_for<Iterators...>{})(iterators...);
}
```

A variadic pack of std::gets

Now that we have the variadic pack of integers, we need to implement `make_transform` in order for it to return a `transform` output iterator containing the `std::get<Is>...`. But we can't just write this:

```
template<size_t... Is>
auto make_transform(std::index_sequence<Is...> const&)
{
    return transform(std::get<Is>...);
}
```

Indeed, `std::get<I>` has 4 overloads: which deal with all four combinations of lvalue/rvalue, and const/non-const inputs. And we can't pass an overloaded function as a parameter, because the compiler doesn't know which overload to choose.

One way to work around this constraint is to group those functions into a function object. And while we're at it, we can group them into a template function inside that function object too, working on any type that `std::get` applies to, so that would include `std::pair`s too.

One way would be to explicitly define a function object, such as:

```
template <size_t I>
struct Get
{
    template <typename Tuple>
    decltype(auto) operator()(Tuple&& tuple)
    {
        return std::get<I>(FWD(tuple));
    }
};
```

`FWD` is a useful macro I learnt from Vittorio Romeo, that alleviates the syntax of calling `std::forward`:

```
#define FWD(value) std::forward<decltype(value)>(value)
```

But Seph De Busser pointed out a more direct way: directly use a variadic pack of lambdas!

```
template<size_t... Is>
auto make_transform(std::index_sequence<Is...> const&)
{
    return transform([](auto&& tup){return std::get<Is>(FWD(tup));}...);
}
```

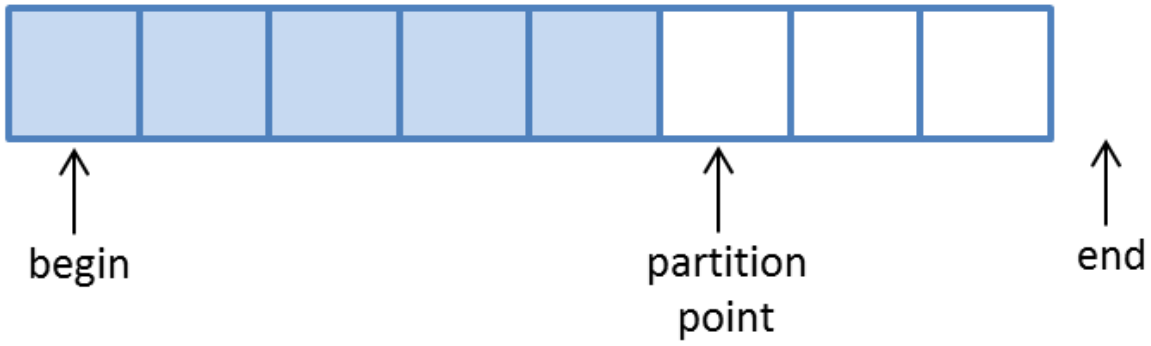
We finally put all this together to create the iterator returned by the `unzip` function:

```
template<typename... Iterators>
auto unzip(Iterators... iterators)
{
    return
make_transform(std::index_sequence_for<Iterators...>{})(iterators...);
}
```

With this new implementation, the unit tests of `unzip` keep passing. Yay!

Partitioning Data with Output Iterators in C++

A couple of months (or years?) back, we saw that partitioning in the STL meant tidying up data according to a predicate: all that satisfy the predicate in one group, and all that don't satisfy the predicate in another group:



This is what the STL algorithms `std::partition` (or `std::stable_partition` to keep the relative order of elements) do:

```
auto numbers = std::vector<int>{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::stable_partition(begin(numbers), end(numbers), [](int n){ return n % 2
== 0; });

for (auto const& number : numbers)
    std::cout << number << ' ';
```

The above program outputs:

```
2 4 6 8 10 1 3 5 7 9
```

All the elements satisfying the predicate are first, the others after.

But there is another way to perform a partition with the STL: putting the values in separate collections. One collection for the elements that satisfy the predicate, and another one for the elements that don't:

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

```
std::partition_copy(begin(numbers), end(numbers),
back_inserter(evenNumbers), back_inserter(oddNumbers), [](int n){ return n
% 2 == 0; });

std::cout << "Even numbers:\n";
for (auto const& number : evenNumbers)
    std::cout << number << ' ';

std::cout << "\nOdd numbers:\n";
for (auto const& number : oddNumbers)
    std::cout << number << ' ';
```

Note that `numbers` is now `const`, since the operation is no longer in place. The outputs are in `evenNumbers` and `oddNumbers` and the above code outputs:

```
Even numbers:
2 4 6 8 10
Odd numbers:
1 3 5 7 9
```

Let's now move that logic out of the algorithm and into the output iterator.

Why a smart output iterator

Before getting into the implementation of an output iterator that performs the equivalent of `std::partition_copy`, why would we want to do such a thing in the first place?

For two reasons:

- breaking off the flow of operations on a collection into two branches,
- chaining additional operations in either or both of those two branches.

To my knowledge, we can't do this with C++ standard components, including with ranges that are coming up in C++20.

Indeed, ranges allow to chain operations, as long as they follow a linear flow:

```
numbers | ranges::view::transform(f) | ranges::view::filter(p);
```

Or they can apply operations that make the data *converge*, that is to say if several sources of data contribute to one result:

```
ranges::view::set_difference(numbers, otherNumbers) |
ranges::view::transform(f);
```

But ranges can't make the data flow diverge, or break off into several directions. This is a key difference between ranges and smart output iterators. They can complete each other, like we'll see in a future post.

We've already seen some smart output iterators, such as `transform` and `filter`:

```
auto const times2 = transform([](int i) { return i*2; });

std::copy(begin(numbers), end(numbers), times2(back_inserter(results));
```

Or, as we'll see in a future post, we can have a nicer syntax:

```
ranges::copy(numbers, transform([](int n){return n*2;}) >>=
back_inserter(results));
```

Or something even nicer by hiding the call to `copy`.

If you hadn't heard of smart output iterators before, you may want to check out this introductory post on smart output iterators, or check out the library on Github.

The partition iterator

Now that we've seen the rationale for implementing a `partition` output iterator, let's decide what we'd like its usage to look like (proceeding this way makes code more expressive):

```
auto const isEvenPartition = partition([](int n){ return n % 2 == 0; });

std::copy(begin(input), end(input),
isEvenPartition(back_inserter(evenNumbers), back_inserter(oddNumbers)));
```

To do this, we'll follow our model for implementing smart output iterators, inspired from one of the most basic smart output iterators out there, the standard `back_inserter`.

We start by implementing `operator*`, that does nothing but returning itself, in order to keep control on the `operator=` that the STL algorithm will typically call afterwards:

```
output_partition_iterator& operator*(){ return *this; }
```

Same thing for `operator++`, not much to do:

```
output_partition_iterator& operator++(){ return *this; }
output_partition_iterator& operator++(int){ ++*this; return *this; }
```

The logic happens in `operator=`. `operator=` receives a value, and needs to send it to either one of the **underlying iterators**, according to whether or not it satisfies the **predicate**.

What follows of the previous sentence is that the iterator has to have access to both its underlying iterators, and to the predicate. We can store them as members in the class, and initialize them in the constructor. The concerned part of the class definition would then look like this:

```
output_partition_iterator(IteratorTrue iteratorTrue, IteratorFalse
iteratorFalse, Predicate predicate)
    : iteratorTrue_(iteratorTrue)
    , iteratorFalse_(iteratorFalse)
    , predicate_(predicate) {}

private:
    IteratorTrue iteratorTrue_;
    IteratorFalse iteratorFalse_;
    Predicate predicate_;
```

Finally, we can implement the `operator=`:

```
output_partition_iterator& operator=(T const& value)
{
    if (predicate_(value))
    {
        *iteratorTrue_ = value;
        ++iteratorTrue_;
    }
    else
    {
        *iteratorFalse_ = value;
        ++iteratorFalse_;
    }
    return *this;
}
```

Matching the desired usage

Remember the desired usage: we wanted to construct the iterator in two phases. First, a function `partition`, that constructed a intermediate object:

```
auto const isEvenPartition = partition([](int n){ return n % 2 == 0; });
```

Then we'd use this object to take the underlying iterators and create the smart iterator we designed above:

```
isEvenPartition(back_inserter(evenNumbers), back_inserter(oddNumbers))
```

We therefore need an intermediary type that takes the predicate in its constructor, and has an `operator()` taking the two underlying iterators to send data to, and returning the `output_parititon_iterator` that we designed.

Let's call this type `output_partitioner`:

```
template<typename Predicate>
class output_partitioner
{
public:
    explicit output_partitioner(Predicate predicate) :
predicate_(predicate) {}
    template<typename IteratorTrue, typename IteratorFalse>
    output_partition_iterator<IteratorTrue, IteratorFalse, Predicate>
operator()(IteratorTrue iteratorTrue, IteratorFalse iteratorFalse) const
    {
        return output_partition_iterator<IteratorTrue, IteratorFalse,
Predicate>(iteratorTrue, iteratorFalse, predicate_);
    }

private:
    Predicate predicate_;
};
```

The `partition` function now just builds an `output_partitioner` (in C++17 with template type deduction in constructors, `partition` could have been the object we called `output_partitioner`):

```
template<typename Predicate>
output_partitioner<Predicate> partition(Predicate predicate)
{
    return output_partitioner<Predicate>(predicate);
}
```

Et voilà le travail!

The whole code is up on Github.

Now we can use `partition` to route the output of an algorithm into two branches, and combine this with other output iterators:

```
auto const isEvenPartition = partition([](int n){ return n % 2 == 0; });
auto const times2 = transform([](int n) { return n*2; });
auto const moreThan3 = filter([](int n) { return n>3; });

ranges::set_difference(input1, input2,
                       isEvenPartition(times2(back_inserter(output1)),
                                       moreThan3(back_inserter(output2))));
```

This code expresses a lot in a few lines, compared to what the version with STL algorithms or for loops would have looked like.

More than two outputs

Our `partition` iterator can split data into two branches according to a predicate. But what if we'd like to split into more than two? What would the interface look like? And the implementation?

This is what we explore in future articles, with the demultiplexer output iterator.