CUSTOM OUTPUT ITERATORS

- **A smart iterator for inserting into a sorted container in C++**
- **A smart iterator for aggregating new elements with existing ones in a map or a set**
- **How the STL inserter iterator really works**
- **How to Use the STL With Legacy Output Collections**
- **Smart Output Iterators: A Symmetrical Approach to Range Adaptors**

Fluent{C++}

A smart iterator for inserting into a sorted container in C++

Smart iterators add great potential to writing expressive code with the STL in C++. And the ones that are proposed natively work particularly well with vectors and with other sequence containers such as deque, list and string.

But the situation is not as good for associative containers, such as maps and sets (or their flat- non-standard counterparts). Indeed, using the native smart iterators is cumbersome and lacks some functionalities. In this 2-post series, I want to propose additions that aim at fixing this situation and letting us write more expressive code when adding elements to an associative container, which is a operation encountered quite frequently in day-to-day code. Of course, your feedback would be very important in the whole process.

To get a grasp of how smart iterators work with the STL, we start by examining `std::back_inserter`, one of those that work well with vectors (if you already know it then you may want to skip the first section, although its case its examined in meticulous details). Then we move on to maps and sets, describe a quick state of the existing standard components, and propose new ones to write expressive code more conveniently.

Appending elements to a vector

`std::back_inserter` generates an output iterator that binds to a container, and does a `push_back` into this container every time it is assigned to. This relieves the programmer from the sizing of the output.

Here is an example of how `std::back_inserter` can be used:

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

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

Here the algorithm `std::copy` assigns elements from `v` to the result of dereferencing the iterator passed via the back_inserter. But `std::back_inserter` generates an iterator that does more than just dereferencing: when you assign through it, it calls a push_back on `results`, passing on the elements of `v` one after one. So that you don't have to worry about `results` being big enough in advance. Smart, right?

We would stop here if it was just about using `std::back_inserter`, but the purpose of this post is to write new smart output iterators. So let's dissect `std::back_inserter` to see what it has in the guts.

First, note that it is not itself an iterator, but rather a function that generates an iterator of type `std::back_insert_iterator`. Since `std::back_insert_iterator` is a template class (templated on the Container), we need a function template to generate it in order to deduce template arguments, otherwise we would have to write them out explicitly at call site (this constraint should be removed in C++17 with template argument deduction for class constructors):

```
template<typename Container>
std::back_insert_iterator<Container> back_inserter(Container& c);
```

So the question is, how does `std::back_inserter_iterator` work? Here is an excerpt of the class where the central thing happens:

```
back_insert_iterator<Container>& operator* () { return *this; }
back_insert_iterator<Container>& operator++ () { return *this; }

back_insert_iterator<Container>& operator= (const typename
Container::value_type& value)
{
    container->push_back(value);
    return *this;
}
```

The iterator binds itself to the container at construction, and dereferencing and advancing do essentially nothing but returning the iterator itself. This has the advantage that the iterator keeps control over `operator=`, to call a push_back on the container.

Adding data to a map

There is a counterpart to `std::back_inserter` to add elements to an `std::map` (or an `std::set`): it is `std::inserter`. Indeed back_inserter cannot be used on a map or a set because they don't have a push_back method. This makes sense: since they guarantee to keep their elements sorted, you can't just decide to puts new elements at the end. So associative containers provide an `insert` method, and `std::inserter` does pretty much the same thing as `std::back_inserter`, except is calls the insert method instead of push_back.

But `std::inserter` shows two flaws when used with maps: it is cumbersome, and it lacks functionality.

Improving usability with sorted_inserter

First, the usability problem: `std::inserter` forces you to give a position where an element should be inserted:

```
template<typename Container>
std::insert_iterator<Container> inserter(Container& c, typename
Container::iterator position);
```

This is all well for a vector, where you *have* to decide for a position. Indeed it could make sense to insert an element anywhere in a vector. But one of the purposes of a map is to be sorted, so the map should take care of deciding where to position a new element, so that it remains sorted! It is certainly not the programmer's job to decide this.

Well, if you happened to know where the new element should be put, then you could save this amount of work to the map, by providing a hint. This is why the insert method of a map has several overloads, including one with a hint parameter:

```
std::pair<iterator,bool> insert(const value_type& value);
iterator                 insert(iterator hint, const value_type& value);
```

But whether or not you provide a hint should be left to the choice of the programmer.

And `std::inserter` **forces you to provide a hint**. But sometimes you don't have a clue. Imagine that you want to add the contents of an unsorted vector into a set. Then you don't have one position where all the elements should go. And we find ourselves passing some arbitrary "hint" because the inserter iterator forces us to, tipically the begin or the end of the set, thus cluttering the code with irrelevant information. Note the unnecessary `results.end()` in he following example:

```
std::vector<int> v = {1, 3, -4, 2, 7, 10, 8};
std::set<int> results;

std::copy(begin(v), end(v), std::inserter(results, end(results)));
```

One solution to fix this is to create a new smart iterator that does essentially the same thing as `std::inserter`, but that does not force its users to provide a hint. Let's call this `sorted_inserter`.

```
template <typename Container>
class sorted_insert_iterator : public
std::iterator<std::output_iterator_tag,void,void,void,void>
{
protected:
  Container* container_;
  boost::optional<typename Container::iterator> hint_;

public:
  typedef Container container_type;
  explicit sorted_insert_iterator (Container& container)
    : container_(&container), hint_(boost::none) {}
  sorted_insert_iterator (Container& container, typename
Container::iterator hint)
    : container_(&container), hint_(hint) {}
  sorted_insert_iterator<Container>& operator= (const typename
Container::value_type& value)
    {
        if (hint_)
            container_->insert(*hint_,value);
        else
            container_->insert(value);
        return *this;
    }
  sorted_insert_iterator<Container>& operator* () { return *this; }
  sorted_insert_iterator<Container>& operator++ () { return *this; }
  sorted_insert_iterator<Container> operator++ (int) { return *this; }
};
```

This iterator can be instantiated with helper functions for deducing template parameters:

```
template <typename Container>
sorted_insert_iterator<Container> sorted_inserter(Container& container)
{
    return sorted_insert_iterator<Container>(container);
}

template <typename Container>
sorted_insert_iterator<Container> sorted_inserter(Container& container,
typename Container::iterator hint)
{
    return sorted_insert_iterator<Container>(container, hint);
}
```

The main difference with `std::inserter` is that **the hint is not mandatory**. This is easily modeled by using an optional (from boost for the moment, from std in C++17). If the hint is provided then we use it, otherwise we let the container decide how to position the inserted element. Note that the operator= taking an r-value reference has been omitted for clarity in this post, but we write by simply replacing the usages of `value` by `std::move(value)`.

Here is how `sorted_inserter` would be used in the above example:

```
std::vector<int> v = {1, 3, -4, 2, 7, 10, 8};
std::set<int> results;

std::copy(begin(v), end(v), sorted_inserter(results));
```

The code for `sorted_inserter` is available on GitHub.

I yet have to benchmark the performance of `std::inserter` versus `sorted_inserter`, to measure whether passing a wrong hint is better or worse than passing none at all. This will likely be the topic of a dedicated post.

This iterator would let you insert new elements in a sorted container. But what if the element you are trying to insert is already present in the container? The default behaviour in the STL is to not do anything. But what if you wanted to **aggregate** the new element with the one already in place? This is the topic of the next post in this series.

A smart iterator for aggregating new elements with existing ones in a map or a set

One thing that is cruelly lacking with `std::inserter` is that it can do just this: inserting. In some situations this is not enough, in particular for a map: what if an element with the same key is already there? `std::inserter`, since it calls `std::map::insert`, will not do anything at all in this case. But maybe we would like to replace the current element with the new one? Or maybe a more complex aggregation behaviour is needed, like adding the values together for example? This last case has been encountered in the project of Coarse Grain Automatic Differentiation when composing derivatives to multiple variables.

This post is part of a series on smart iterators in sorted containers:

- `sorted_inserter`: A smart iterator for inserting into a map or any sorted container
- `map_aggregator`: A smart iterator for aggregating a new element with an existing one into a map or a set

Said differently, we need a even smarter iterator, to which you could describe what to do when trying to insert elements with keys already present in the map. The most generic expression I found was to provide an **aggregator**, that is, a function that describes how to merge two values, for elements having the same key. This would let new elements being "inserted" into the map regardless of whether or not their key is already present, and still keep the unicity of the key in the map (so this solution is effectively different from using a multimap).

Here is how `map_aggregator` could be implemented:

```
template<typename Map, typename Function>
class map_aggregate_iterator : public
std::iterator<std::output_iterator_tag, void, void, void, void>
{
public:
```

```
    map_aggregate_iterator(Map& map, Function aggregator) : map_(map),
aggregator_(aggregator) {}
    map_aggregate_iterator operator++(){ return *this; }
    map_aggregate_iterator operator*(){ return *this; }
    template<typename KeyValue>
    map_aggregate_iterator& operator=(KeyValue const& keyValue)
    {
        auto position = map_.find(keyValue.first);
        if (position != map_.end())
        {
            position->second = aggregator_(position->second,
keyValue.second);
        }
        else
        {
            map_.insert(position, keyValue);
        }
        return *this;
    }

private:
    Map& map_;
    Function aggregator_;
};
```

Here is a helper function to instantiate it and deduce template parameters:

```
template<typename Map, typename Function>
map_aggregate_iterator<Map, Function> map_aggregator(Map& map, Function
aggregator)
{
    return map_aggregate_iterator<Map, Function>(map, aggregator);
}
```

This has several major differences with `std::inserter`:

- `map_aggregator` embarks a aggregator function in its constructor,
- `operator=` aggregates the new value into the existing element by using the aggregator function, if the key is already present in the collection.
- Like `sorted_inserter` presented in the previous post of this series, you don't have to pass a hint. (In fact you could pass it if you knew it, but to alleviate the code in this post I am not showing this functionality here.)

Here is a how `map_aggregator` can be used:

```
std::vector<std::pair<int, std::string>> entries = { {1, "a"}, {2, "b"},
{3, "c"}, {4, "d"} };
```

```
std::vector<std::pair<int, std::string>> entries2 = { {2, "b"}, {3, "c"},
{4, "d"}, {5, "e"} };
std::map<int, std::string> results;

std::copy(entries.begin(), entries.end(), map_aggregator(results,
concatenateStrings));
std::copy(entries2.begin(), entries2.end(), map_aggregator(results,
concatenateStrings));

// results contains { {1, "a"}, {2, "bb"}, {3, "cc"}, {4, "dd"}, {5, "e"}
};
```

Here the first call to `map_aggregator` is not strictly necessary, since the collection `results` is empty. It could be replaced by an simple `std::inserter` or, more to the point, by a `sorted_inserter` presented in the first post of this series.

What about sets?

The above aggregator has been designed to work with maps, that contain pairs of keys and values. But sometimes the key of an element is embedded inside the element, like with a reference number that is a member of an object for example. In this case you may want to use a set with a customized comparison based on the subpart of the element that represents the key.

We can then define another smart iterator for aggregating into a set, with much the same logic as the one for maps, the main difference lying in the `operator=`:

```
set_aggregate_iterator& operator=(Value const& value)
{
    auto position = set_.find(value);
    if (position != set_.end())
    {
        auto containedValue = *position;
        position = set_.erase(position);
        set_.insert(position, aggregator_(value, containedValue));
    }
    else
    {
        set_.insert(position, value);
    }
    return *this;
}
```

The thing with sets is that they don't allow their values to be modified (some platforms let you get away with it but relying on that prevents code from being portable). Therefore, we have to remove the old value and then add the aggregated one. This is what `operator=` does here when it finds that the element was there already.

For clarity, the rest of the implementation of this inserter in sets is omitted in the writing of this post, but it's essentially the same as the one for maps.

To see the entire code of the components presented here, you can head over to the dedicated GitHub repository.

Over to you

Do you find these components of this series useful? Have you encountered the problems they are solving? How would you have gone about solving them differently?

Whether you are a new reader on Fluent C++ or a regular one, **your feedback matters to me**. And not only on this particular series by the way. Depending on the size and visibility you want your feedback to have you can drop a comment below, or use email or Twitter to get in touch directly. Hoping to hear from you!

How the STL inserter iterator really works

The inserter iterators such as `std::back_inserter` and `std::inserter` are important components in the STL that participate in letting us improve the expressiveness of our code.

Here we delve into `std::inserter`. We'll start with a basic question concerning how it can work, have a peek at the inside, and answer that question. This will make us better understand what it *really* does.

Special thanks to my colleague Gabriel Fournier with whom we scratched our heads dangerously hard over this, and who's always a source of valuable feedbacks and ideas.

How can the thing work?

Quoting cppreference.com, the object generated by `std::inserter` "inserts elements into a container for which it was constructed, at the position pointed to by the supplied iterator". Here is how it looks like in code:

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

std::copy(begin(newElements), end(newElements), std::inserter(v, v.end()));

for (int i : v) std::cout << i << ' ';
```

The above code aims at appending the elements in `newElements` at the end of `v`.

(Actually `std::inserter` really becomes useful for more complex uses. For such a basic use as the one above you'd rather blast the elements at the end using a range insertion method on `vector`. But here let's consider the simplest usage to better demonstrate how `std::inserter` works. And if you want to know how to to make efficient insertions, you can read Inserting several elements into an STL container efficiently).

But how can this work? `std::inserter` takes a position inside the container, and make repeated insertions. So, at some point we would expect the buffer inside the vector to overgrow, causing an reallocation and an invalidation of the position passed to `std::inserter`, right? And even if the buffer was large enough so as not to be reallocated, repeatedly inserting at a position should produce the elements in reverse order, shouldn't it?

If these suspicions turn out to be founded, then we would have a spectacular bug in the STL. Quick, let's test this and see if we got this right. Compile, execute, and the output is…

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

Yeah well, the STL works. Duh. Now the question is: how?

What it looks like it does

Let's try to simulate the behaviour we were assuming `std::inserter` had. We take a position inside a container, and repeatedly insert elements at this position:

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

auto position = v.end();
for (int i : newElements) v.insert(position, i);

for (int i : v) std::cout << i << ' ';
```

The following code outputs:

```
Segmentation fault
```

Just as we expected. The vector's buffer is reallocated and `position` is invalidated. The next insertion through it leads to undefined behaviour.

Let's try to work around this by reserving buffer space in advance:

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

v.reserve(10);
auto position = v.end();
for (int i : newElements) v.insert(position, i);
```

```
for (int i : v) std::cout << i << ' ';
```

With this the program now longer crashes, but we get the following output:

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

The inserted elements are in reverse order because, as we predicted, repeatedly inserting at the same position pushes the last insertions to the right every time.

What it actually does

So how does `std::inserter` do to insert the elements in the right order without crashing? **How does it just do the right thing**?

Let's look into an implementation of the STL to understand what's going on. Here is the code that actually performs the insertion of an element inside the container:

```
_Self& operator=(const typename _Container::value_type& __val) {
  _M_iter = container->insert(_M_iter, __val);
  ++_M_iter;
  return *this;
}
```

`_M_iter` is the position that was passed to `std::inserter`. As you can see, there is more than just an insertion:

- the position is **refreshed** by using the one returned by the `insert` method. `insert` returns the position of the inserted element. This ensures to always keep `_M_iter` in the vector's current buffer, even if it was just reallocated.
- the position is incremented. This way each inserted element is positioned right **after** the previous one. This ensures to have the elements inserted in the right order.

So here was `std::inserter` mystified and then demystified. Hope you appreciated the little journey, and learned a thing or two along the way. We sure did.

How to Use the STL With Legacy Output Collections

When you start using the STL and its algorithms in your code, it's a bit of a change of habits. And then after a while you get used to it. Then it becomes a second nature. And then even your dreams become organized into beautifully structured ranges that fly in and out of well-oiled algorithms.

And when you reach that point, there is no coming back.

Until the day you come upon an old legacy structure that won't let itself approached by the elegant and expressive way of coding that STL algorithms have. It's a terrible encounter, where the beast tries to suck you back into the lengthy and dangerous quicksand of the raw for loops that now seemed so far away.

I've faced that day with my valiant colleague Gauthier, and together we drove an epic fight until we forced the beast into a several-inch thick STL prison, where it could no longer harm the rest of the code. Ok, it wasn't *that* epic. But anyway, let me tell you that tale so that you can use it if you face a similar situation. We'll see the main component that allowed us to do this, `custom_inserter`, so that you don't need to dress up for this fight again (I later realized that something very close existed in Boost, boost function output iterator, so you'll prefer that if you can use Boost in your code).

In other words, let's see **how to use the STL algorithms with legacy input and outputs**.

We've already touched upon legacy or user-defined **inputs**, by studying the design of the STL. So now we'll focus on how to **output** the results of an algorithm into a legacy structure that wasn't designed to be compatible with the STL.

The case

I'm going to simplify the use case to the bare minimum to spend the less amount of time understanding it.

We have a collection of inputs, say in the form of a `vector`:

```
std::vector<Input> inputs = //...
```

and a function `f` that we want to apply to each one of them:

```
Output f(Input const& input);
```

This will result into as many `Output`s. And we need to feed these outputs to an object that isn't an STL container, and that doesn't look like one. Maybe it's an old C `struct`, or maybe it's something more complicated. We'll call this object `legacyRepository`, of type `LegacyRepository`. That's the beast.

And `legacyRepository` comes with a function to add things into it:

```
void addInRepository(Output const& value, LegacyRepository&
legacyRepository);
```

It doesn't have to be of that particular form, but I'm choosing this one to illustrate, because it really doesn't look like STL containers' typical interface.

If we could replace the old repository by an `std::vector`, then we'd have used `std::transform` with `std::back_inserter` and be done with it:

```
std::transform(begin(inputs), end(inputs), std::back_inserter(repository),
f);
```

But you can't always refactor everything, and in this case we couldn't afford to refactor this right now. So, how should we proceed?

A generalisation of std::back_inserter

I think we should take inspiration from `std::back_inserter` that outputs into a vector, in order to create a generalized component that can output into anything.

From this point on and until the end of this section I'm going to show you the reasoning and implementation that went into the component, `custom_inserter`. If you only want the resulting component then you can just hop on to the next section.

So, how does `std::back_inserter` works? It creates an output iterator, `std::back_insert_iterator`, that features the two required methods `operator++` and `operator*`. But the real point of `std::back_inserter` is to take control on how the new values are assigned into the container it is linked to, and it does so with its `operator=`:

```
back_insert_iterator& operator=(T const& value)
{
    container_.push_back(value);
    return *this;
}
```

(This code wasn't taken from any STL implementation, it is theoretical code to illustrate what `std::back_inserter` is doing.)

But then, how come it's the `operator=` of `std::back_insert_iterator` that is called, and not the `operator=` of the type inside the collection? It's because `operator*` doesn't return an element of the collection, it rather keeps the control in the smart iterator:

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

And `operator++` must be implemented but doesn't play a role in all this, so it is pretty much reduced to a no-op:

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

This technique works well on containers that have a `push_back` method, but why not use the same mechanism for containers that have another interface?

custom_inserter

So let's create our `custom_insert_iterator`, that, instead of taking a container, takes a custom function (or function object) to replace the call to `push_back`:

```
template<typename OutputInsertFunction>
class custom_insert_iterator
{
public:
    using iterator_category = std::output_iterator_tag;
    explicit custom_insert_iterator(OutputInsertFunction insertFunction) :
insertFunction_(insertFunction) {}
    custom_insert_iterator& operator++(){ return *this; }
    custom_insert_iterator& operator*(){ return *this; }
    template<typename T>
    custom_insert_iterator& operator=(T const& value)
    {
        insertFunction_(value);
        return *this;
    }
private:
    OutputInsertFunction insertFunction_;
};
```

And the `custom_inserter` helper function to avoid to specify template parameters at call site:

```
template <typename OutputInsertFunction>
custom_insert_iterator<OutputInsertFunction>
custom_inserter(OutputInsertFunction insertFunction)
{
    return custom_insert_iterator<OutputInsertFunction>(insertFunction);
}
```

Here is how we can use it:

```
std::copy(begin(inputs), end(inputs),
    custom_inserter([&legacyRepository](Output const&
value){addInRepository(value, legacyRepository);}));
```

If you find this expression too cumbersome we can abstract the lambda:

```
auto insertInRepository(LegacyRepository& legacyRepository)
{
    return [&legacyRepository](Output const& value)
    {
        addInRepository(value, legacyRepository);
    };
}
```

in order to have a simpler call site:

```
std::transform(begin(inputs), end(inputs),
custom_inserter(insertInRepository(legacyRepository)));
```

Couldn't it be simpler?

As underlined by Nope in the comments section, this illustration is pretty simple and could be worked around with a simple code like:

```
for (const auto& input: inputs) addInRepository(f(input),
lecgacyRepository);
```

Even though this code declares an `input` variable that is not necessary to express the idea of "applying `f` on the collection", the above line of code is simpler than using a `custom_inserter`.

`custom_inserter` becomes really helpful to leverage on more elaborate STL algorithms, for example on the algorithms on sets:

```
std::set_difference(begin(inputs1), end(inputs1),
                    begin(inputs2), end(inputs2),
                    custom_inserter(insertInRepository(legacyRepository)));
```

Is this more or less legacy?

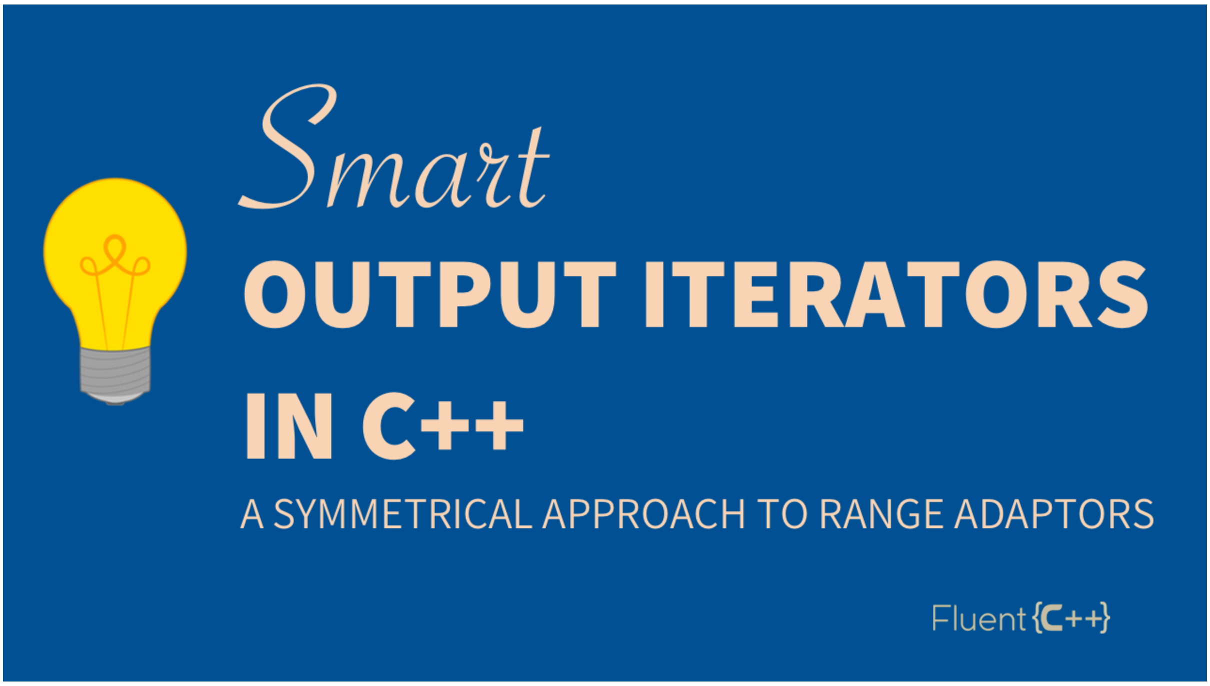
One could argue that we didn't reduce the amount of legacy, because `LegacyRepository` hasn't changed a bit, but a new non-standard component (or the one from Boost) has appeared on the top of it. So is it worth it?

I think we should weigh our other options in that situation. If we can get rid of the legacy and have a nice `vector`, or an otherwise STL-compatible interface instead (that is, that has at least a `push_back` method), then by all means we should do it. This way we'd have STL code all the way, and standard components to insert into the collection. This is the most desirable situation.

But if we can't, or if it isn't realistic on this particular piece of code (maybe it would take months or years to take down, or maybe this is an external API and we just don't have control over it), the way I see it is that we're facing two options: forgoing the usage of STL algorithms on this piece of code, with all the implications that we know, or using STL algorithms with our non-standard `custom_inserter`, which is not ideal because it is not standard, and it has a level of indirection. And next time you face this situation in your code, you'll have to make a choice.

In all cases, `custom_inserter` is there for you, and don't hesitate to give your feedback if you have any.

Smart Output Iterators: A Symmetrical Approach to Range Adaptors



Some of the algorithms of the STL have a structure in common: they take one or more ranges in input, do something more or less elaborate with them, and produce an output in a destination range.

For example, `std::copy` merely copies the inputs to the outputs, `std::transform` applies a function onto the inputs and sends the results as outputs, and `std::set_difference` takes two input ranges and outputs to a destination range the elements that are in the first one but not in the second.

There are several ways to express this kind of input-operation-output structure on ranges in C++. To illustrate them, let's take the example of `std::transform` since it is such a central algorithm in the STL.

To make the code examples lighter, let's suppose that we have some modified versions of STL algorithms that take an input range instead of two iterators, for instance:

```
namespace ranges
{
template <typename Range, typename OutputIterator>
OutputIterator copy(Range const& range, OutputIterator out)
{
    return std::copy(range.begin(), range.end(), out);
}
}
```

and so on for other algorithms.

Various places to put the logic

The standard way to apply a function to each element and have the results added to a collection is to combine the `std::transform` algorithm with an output iterator such as `std::back_inserter`:

```
// f is a function to apply to each element of the collection
int f(std::string const& s);

std::vector<std::string> strings = { "So", "long", "and", "thanks", "for",
"all", "the", "fish" };
std::vector<int> results;

ranges::transform(strings, std::back_inserter(results), f);
```

A more modern way, which logic we saw in Ranges: the STL to the Next Level, is to use ranges and range adaptors:

```
// f is a function to apply to each element of the collection
int f(std::string const& s);

std::vector<std::string> strings = { "So", "long", "and", "thanks", "for",
"all", "the", "fish" };
std::vector<int> results;

ranges::copy(strings | ranges::view::transform(f),
std::back_inserter(results));
```

We could even do away with the back_inserter here by using the `push_back` free function, but let's keep it generic to take account of the case of sending outputs to a stream for example.

One interesting thing to note here is that the main action of the whole operation, which is applying the function `f`, has been transferred to the input range: `strings | ranges::view::transform`, taking this responsibility away from the algorithm. The algorithm then becomes simpler, becoming `copy` instead of `transform`.

When we see it from this perspective, we can see another way of structuring the operation. One that gets less publicity than the other ones, but that can have several advantages as we'll see in just a moment: shifting the logic to the output iterator:

```
// f is a function to apply to each element of the collection
int f(std::string const& s);

std::vector<std::string> strings = { "So", "long", "and", "thanks", "for",
"all", "the", "fish" };
std::vector<int> results;

ranges::copy(strings, transform_f(std::back_inserter(results)));
```

where `transform_f` is an output iterator that applies f and forwards this result to the `std::back_inserter`.

Note that with this approach the input range is simple (`strings`), the algorithm is simple too (`ranges::copy`) and the responsibility of applying `f` has been moved over to the output iterator.

Is this form helpful at all?

The case for smart output iterators

Let's take a case where standard algorithms aren't practical to use: the case of "transform if" for example. This is a case where we'd like to apply a function to only the elements of a collection that satisfy a predicate. It is cumbersome to do with the STL because STL algorithms don't chain up well:

```
int f(int);

std::vector<int> numbers = {1, 2, 3, 4, 5};
std::vector<int> evenNumbers;
copy_if(numbers, std::back_inserter(evenNumbers), isEven);
std::vector<int> results;
transform(evenNumbers, std::back_inserter(results), f);
```

So let’s say that the first way using STL algorithms is out. We’re left with two options:

using ranges:

```
int f(int);

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

ranges::copy(numbers | ranges::view::filter(isEven) |
ranges::view::transform(f), std::back_inserter(results);
```

using smart output iterators:

```
int f(int);

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

ranges::copy(numbers,
filter_even(transform_f(std::back_inserter(results))));
```

Smarter output iterators

Ranges are more and more the default solution in this case, and the direction that the STL is taking for the future. However, there are several reasons why it can be interesting to consider giving some responsibility to output iterators.

The first reason is that for the algorithms taking more than one range in input, for example `std::set_difference` and the other algorithms on sets, to my knowledge you can’t use traditional range adaptors to apply a transformation to the outputs of the algorithms. Indeed, ranges adaptors could modify either one or both of the input ranges:

```
set_difference(range1 | adaptor1,
               range2 | adaptor2,
               outputIterator);
```

But how could they apply a transformation on the outputs of the algorithms before sending them to the `outputIterator`, like a smart output iterator would do?

EDIT: in fact, the STL algorithms on sets are not such a good example of absolute necessity for smart output iterators, since range-v3 turns out to *have* view adaptors on sets algorithms. But there are still other cases where they are necessary, for instance algorithms that have several outputs. The STL only has `std::partition_copy`, but it's very useful to extend the STL with more elaborate algorithms such as `set_segregate`, which has multiple outputs. In this case, smart output iterators become very handy.

A second reason is that smart output iterators could better express that some transformations are not semantically related to the algorithm, but rather to how the output collection stores its elements. To illustrate, let's consider the case where the output container stores BigInts instead of ints. And this BigInt class doesn't allow implicit conversion because its designer was wary of implicit conversions.

So our function `f` here would convert an int into a BigInt, simply by calling its constructor:

```
BigInt make_bigint(int i)
{
    return BigInt(i);
}
```

In this case, when reading the code we don't really care about the fact that `f` is called. It has to be there, otherwise the code wouldn't compile, but the **meaningful** part in the code is arguably the application of the predicate `isEven`. Shifting this application of `f` to the output iterator is a way to convey this message: this is just to make the outputs fit into the output container, much like `std::back_inserter` is.

So we could delegate the responsibility of the conversion to the output iterator side and mix both ranges and output iterators:

```
int f(int);

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

ranges::copy(numbers | ranges::view::filter(isEven),
```

```
                bigint_convert(std::back_inserter(results)));
```

or we could just use the STL algorithm, here `copy_if`:

```
int f(int);

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

ranges::copy_if(numbers,
                bigint_convert(std::back_inserter(results)),
                isEven);
```

Another reason is a very practical one: smart output iterators are light-weight components that are relatively easy and quick to implement (much easier than ranges, I've tried to implement both) even in C++03. We see an example of that in the next section. So if you don't have access to Boost Ranges or range-v3, they can be a **practical way** to make your code more concise. We'll see an implementation in the next section of this article.

Finally, a last reason to consider smart output iterators is that they are a **different way** to go about structuring the call to an algorithm. And just for that reason, they can expand our view and give us more perspective on the topic of applying algorithms!

Implementing smart output iterators

To follow up on the above example with BigInt, let's make a generic output iterator that takes a function, applies it to the value it receives, and sends the result on to the iterator that it wraps (a `std::back_inserter` for example).

Here is a complete implementation, that we detail bit by bit just after:

```
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_;
};

template<typename TransformFunction>
class output_transformer
{
public:
    explicit output_transformer(TransformFunction transformFunction) :
transformFunction_(transformFunction) {}
    template<typename Iterator>
    output_transform_iterator<Iterator, TransformFunction>
operator()(Iterator iterator) const
    {
        return output_transform_iterator<Iterator,
TransformFunction>(iterator, transformFunction_);
    }

private:
    TransformFunction transformFunction_;
};

template<typename TransformFunction>
output_transformer<TransformFunction>
make_output_transformer(TransformFunction transformFunction)
{
    return output_transformer<TransformFunction>(transformFunction);
}
```

Here is how this code works:

The generic elements of the smart iterator are:

- the function to apply,
- the iterator it wraps.

So let's makes these two template parameters:

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

Let's accept those two parameters in the constructor and store them in our smart iterator:

```
output_transform_iterator(Iterator iterator, TransformFunction
transformFunction) : iterator_(iterator),
transformFunction_(transformFunction) {}

private:
    Iterator iterator_;
    TransformFunction transformFunction_;
```

We need to implement the operators of an output iterator: `operator++` advances the underlying iterator. Advancing the underlying iterator is a no-op in `std::back_inserter`, but is necessary if the underlying output iterator is the `begin` of a container for example.

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

And like for `std::back_inserter` and `custom_inserter`, we use `operator*` to return the iterator itself and keep control of `operator=` to apply the function and pass the result to the underlying iterator:

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

That's about it, except that the interface isn't quite right: we would like an iterator that wraps over another iterator, and not one that also takes a function in its constructor:

```
bigint_convert(std::back_inserter(results))
```

Said differently, we'd like to partially apply the constructor with the transform function, here `make_bigint`, retrieve the object and give it an underlying iterator at a later time.

To simulate partial function application of a function in C++, we can use a function object:

```
template<typename TransformFunction>
class output_transformer
{
public:
    explicit output_transformer(TransformFunction transformFunction) :
transformFunction_(transformFunction) {}
    template<typename Iterator>
```

```
    output_transform_iterator<Iterator, TransformFunction>
operator()(Iterator iterator) const
    {
        return output_transform_iterator<Iterator,
TransformFunction>(iterator, transformFunction_);
    }

private:
    TransformFunction transformFunction_;
};
```

Indeed, the parameters are applied in two phases: the first one in the constructor and the second one in the `operator()`.

Finally, to create a `transformer` we use a helper function to deduce the template parameter of the transform function:

```
template<typename TransformFunction>
output_transformer<TransformFunction>
make_output_transformer(TransformFunction transformFunction)
{
    return output_transformer<TransformFunction>(transformFunction);
}
```

This implementation is compatible with C++03 (and I didn't see how to use lambdas to make it clearer anyway). Note though that in C++17 we wouldn't need the `make_output_transformer` function thanks to the type deduction in class template constructors.

Sweeping low-level operations under the rug

By using the smart output iterator we can now make the conversion to BigInt more discrete at the call site:

```
//C++03
output_transformer<BigInt(*)(int)> const bigint_converter =
make_output_transformer(make_bigint);

//C++11
auto const bigint_converter = make_output_transformer(make_bigint);

//C++17
```

```cpp
auto const bigint_converter = output_transformer(make_bigint);

int f(int);

//Call site
std::vector<int> numbers = {1, 2, 3, 4, 5};
std::vector<BigInt> results;

ranges::copy(numbers | ranges::view::filter(isEven),
             bigint_convert(std::back_inserter(results)));
```

Will smart output iterators compete with ranges on all use cases? Certainly not. But to express that an operation is more closely related to the output container than to the algorithm itself, they can constitute an alternative worth having in our toolbox.

`output_transformer` and other smart output iterators are available in the smart-output-iterators (now renamed pipes) GitHub repository.