Smart OUTPUT ITERATORS IN C++

A SYMMETRICAL APPROACH TO RANGE ADAPTORS

Jonathan Boccara

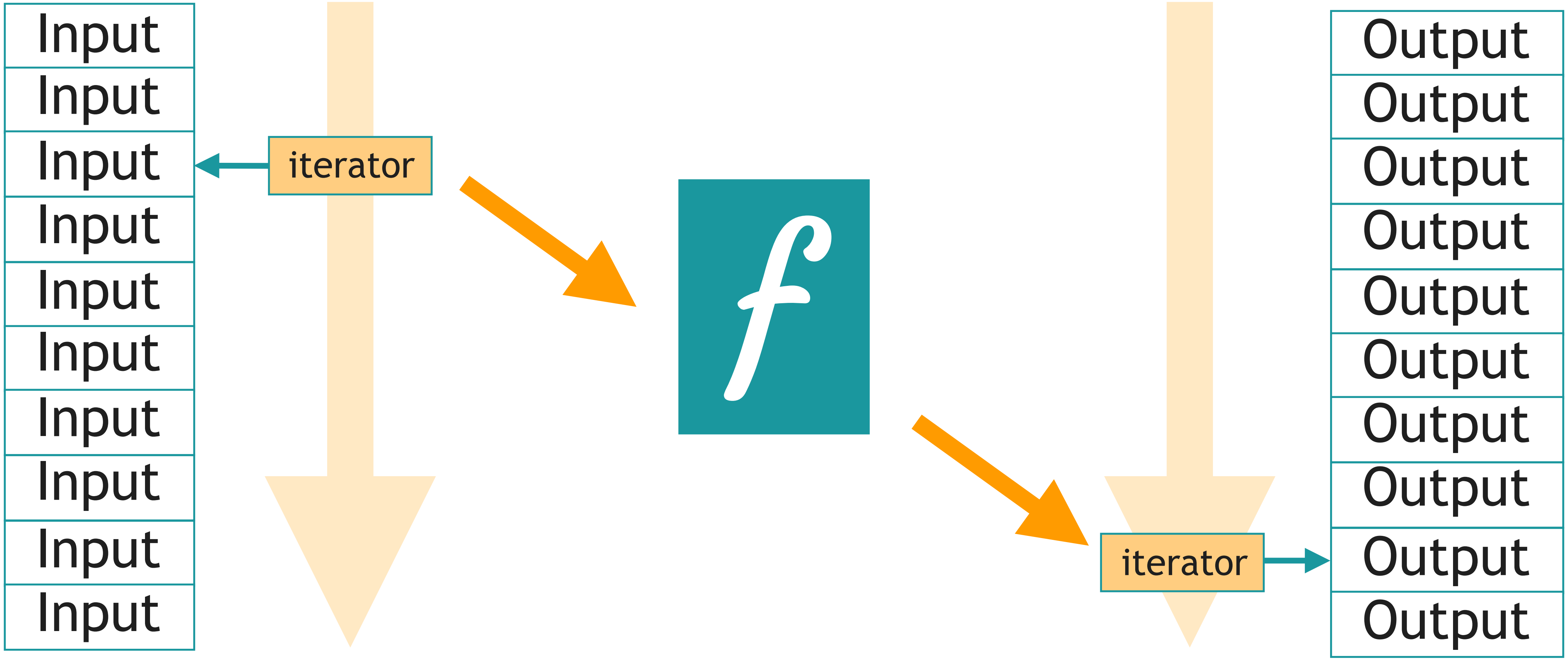
Fluent{C++}

MUREX™ Financial Software

APPLYING A FUNCTION

APPLYING A FUNCTION ON A COLLECTION

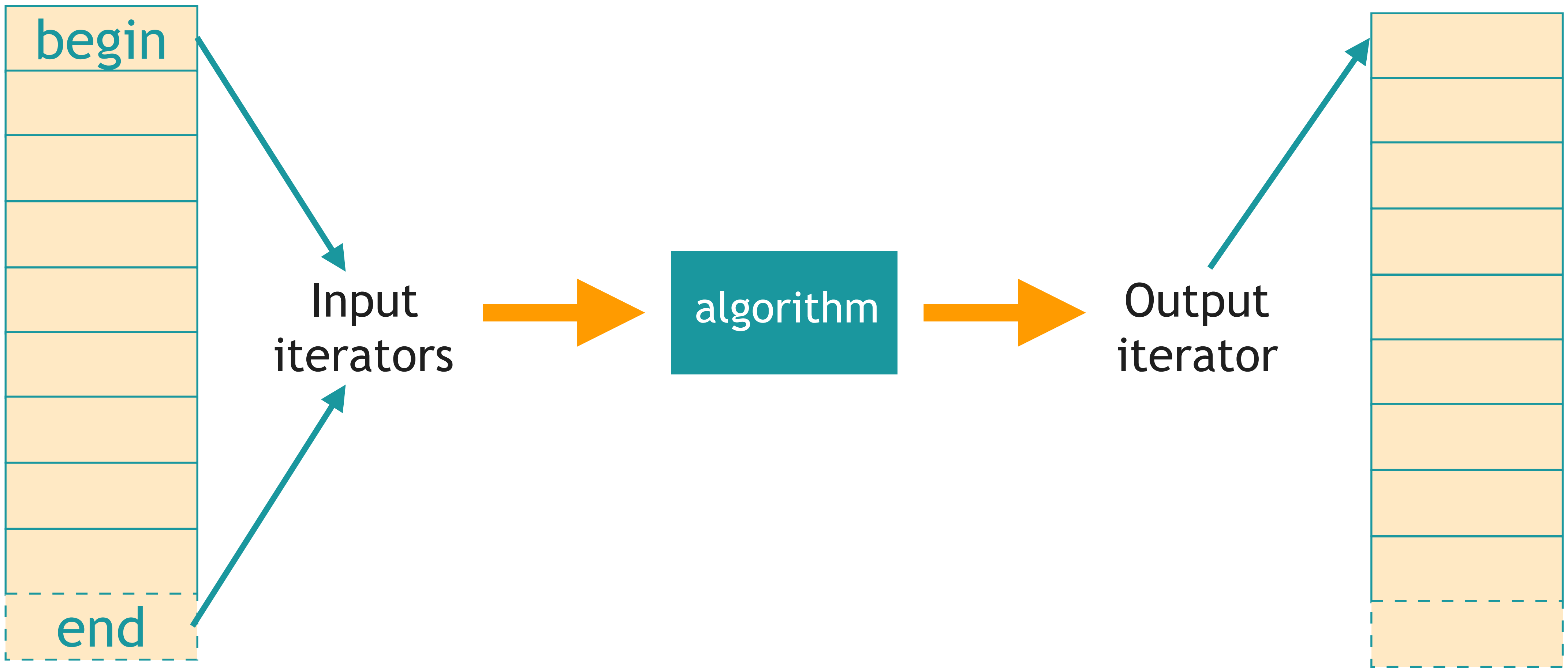
APPLYING A FUNCTION ON A COLLECTION



APPLYING A FUNCTION

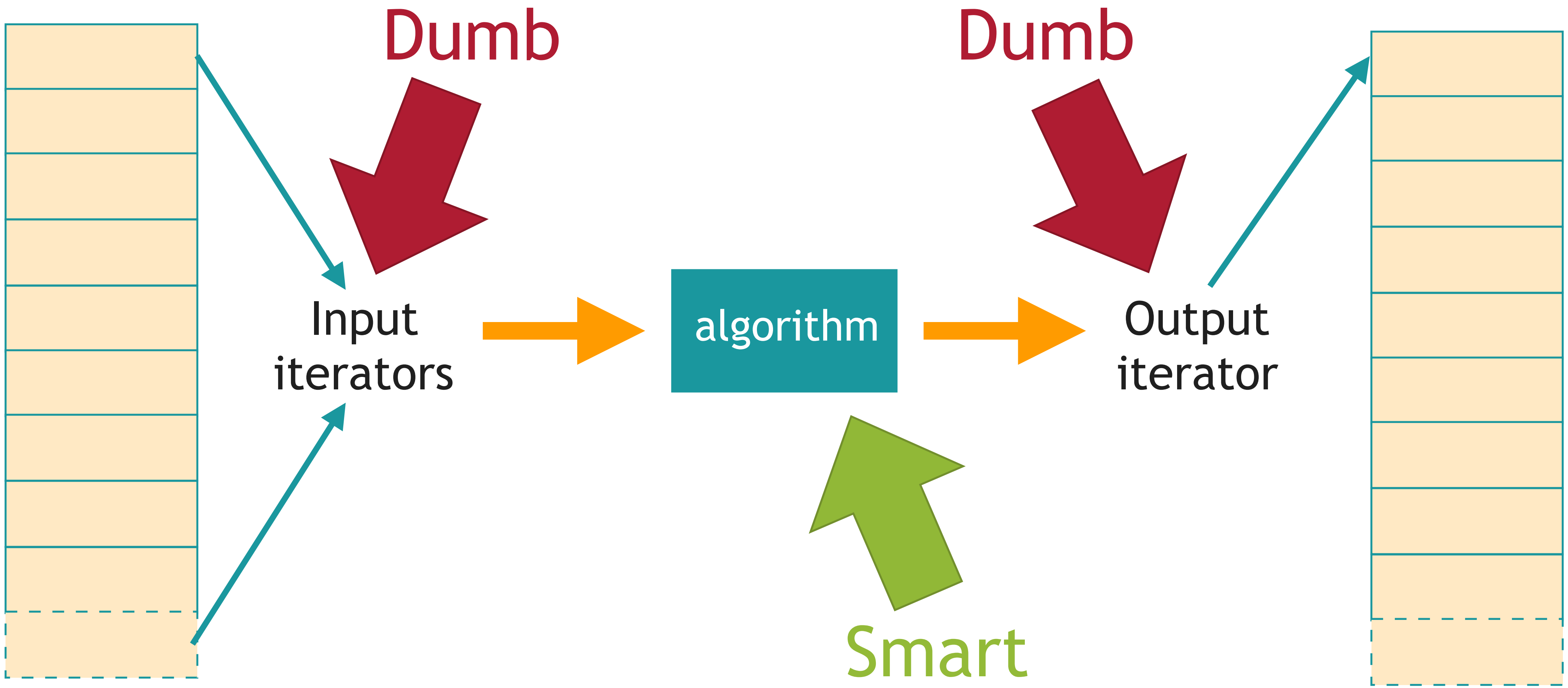
APPLYING A FUNCTION

THE DESIGN OF THE STL



```
template<template InputIterator, template OutputIterator, template Function>
OutputIterator transform(InputIterator first, InputIterator last, OutputIterator out, Function f);
```

THE DESIGN OF THE STL



SMART ALGORITHMS

```
template<typename InputIterator, typename OutputIterator, typename Function>
OutputIterator transform(InputIterator first, InputIterator last, OutputIterator out, Function func)
{
    while (first != last) {
        *out++ = func(*first++);
    }
    return out;
}

template<typename InputIterator, typename OutputIterator, typename UnaryPredicate>
OutputIterator copy_if(InputIterator first, InputIterator last, OutputIterator out, Predicate pred)
{
    while (first != last) {
        if (pred(*first))
            *out++ = *first;
        first++;
    }
    return out;
}
```

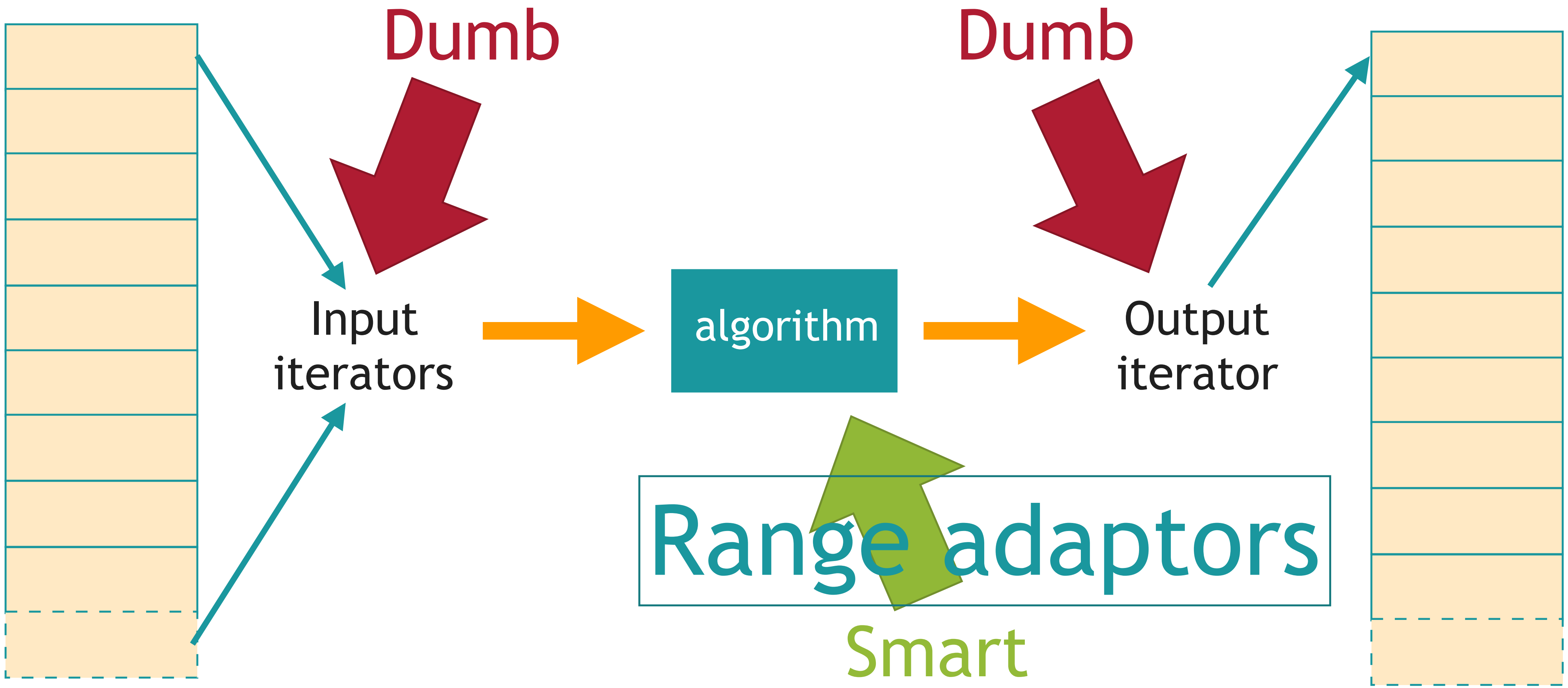
SMART ALGORITHMS

```
template<typename InputIterator1, typename InputIterator2, typename OutputIterator>
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                              InputIterator2 first2, InputIterator2 last2,
                              OutputIterator out)
{
    while (first1 != last1) {
        if (first2 == last2) return std::copy(first1, last1, out);

        if (*first1 < *first2) {
            *out++ = *first1++;
        } else {
            if (! (*first2 < *first1)) {
                ++first1;
            }
            ++first2;
        }
    }
    return out;
}
```

DUMB ITERATORS

++ → Move one step forward

* → Give a reference to current element



MURODEX
Financial Software

RANGE ADAPTORS

Smart input iterators

```
bool isEven(int);
int times2(int);

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

copy(numbers | view::filter(isEven) | view::transform(times2),
     std::back_inserter(results));
```

FILTER ITERATOR

`view::filter(pred)`

++ → Move to the next element that satisfies pred

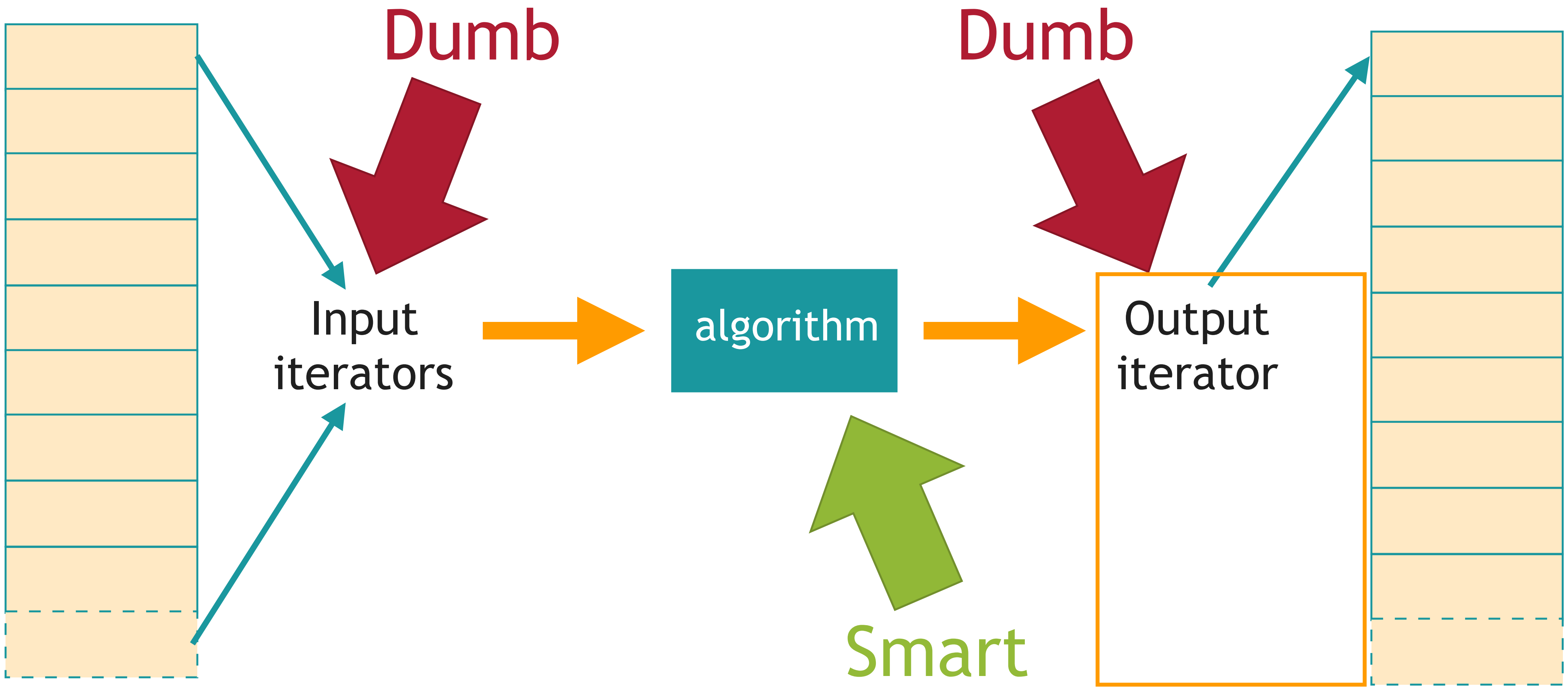
* → Give a reference to current element

TRANSFORM ITERATOR

`view::transform(func)`

++ → Move one step forward

* → Give the value returned by func(element)


Dumb
Dumb
Input iterators
algorithm
Output iterator
Smart

Smart
OUTPUT ITERATORS IN C++

A SYMMETRICAL APPROACH TO RANGE ADAPTORS

Jonathan Boccara

Fluent{C++}

MUREX™ Financial Software

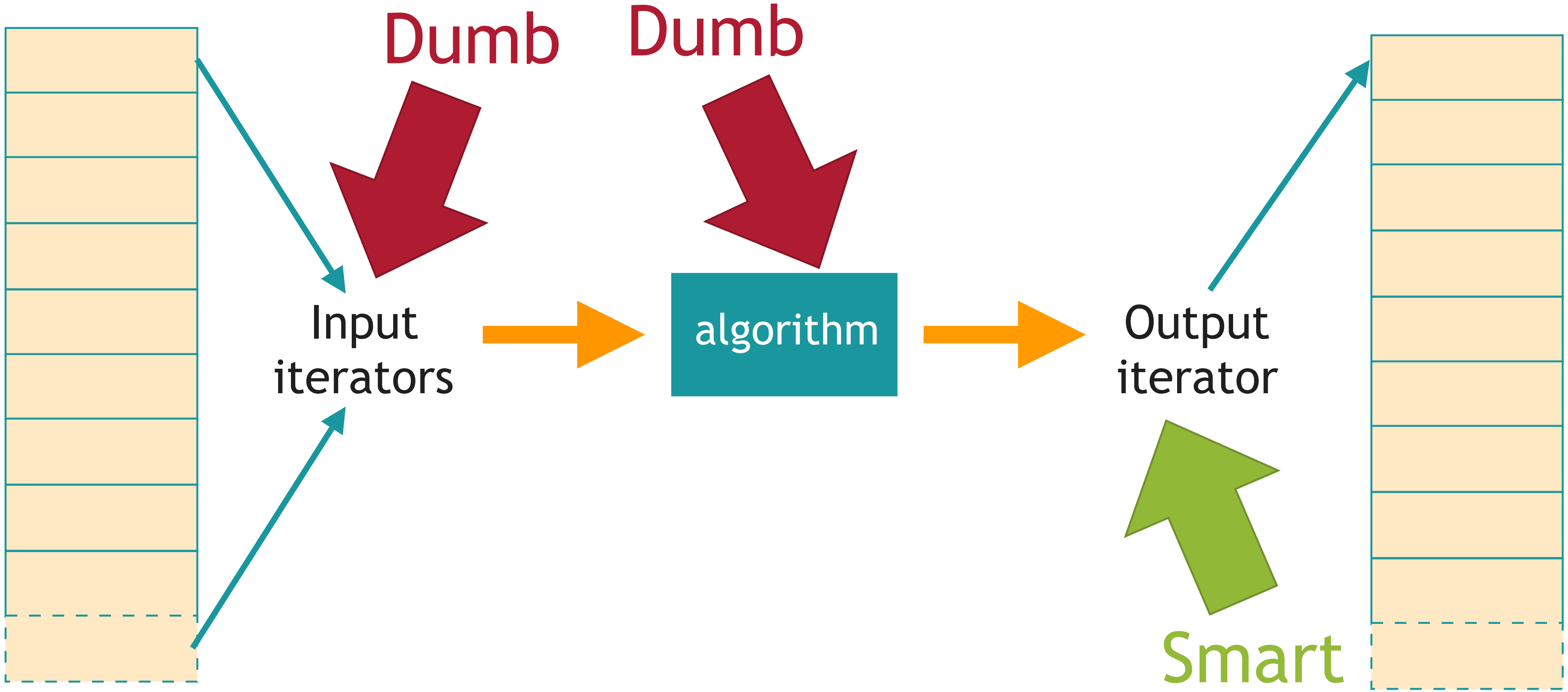
Hi, I’m Jonathan Boccara!

@JoBoccara

EXPRESSIVE CODE IN C++

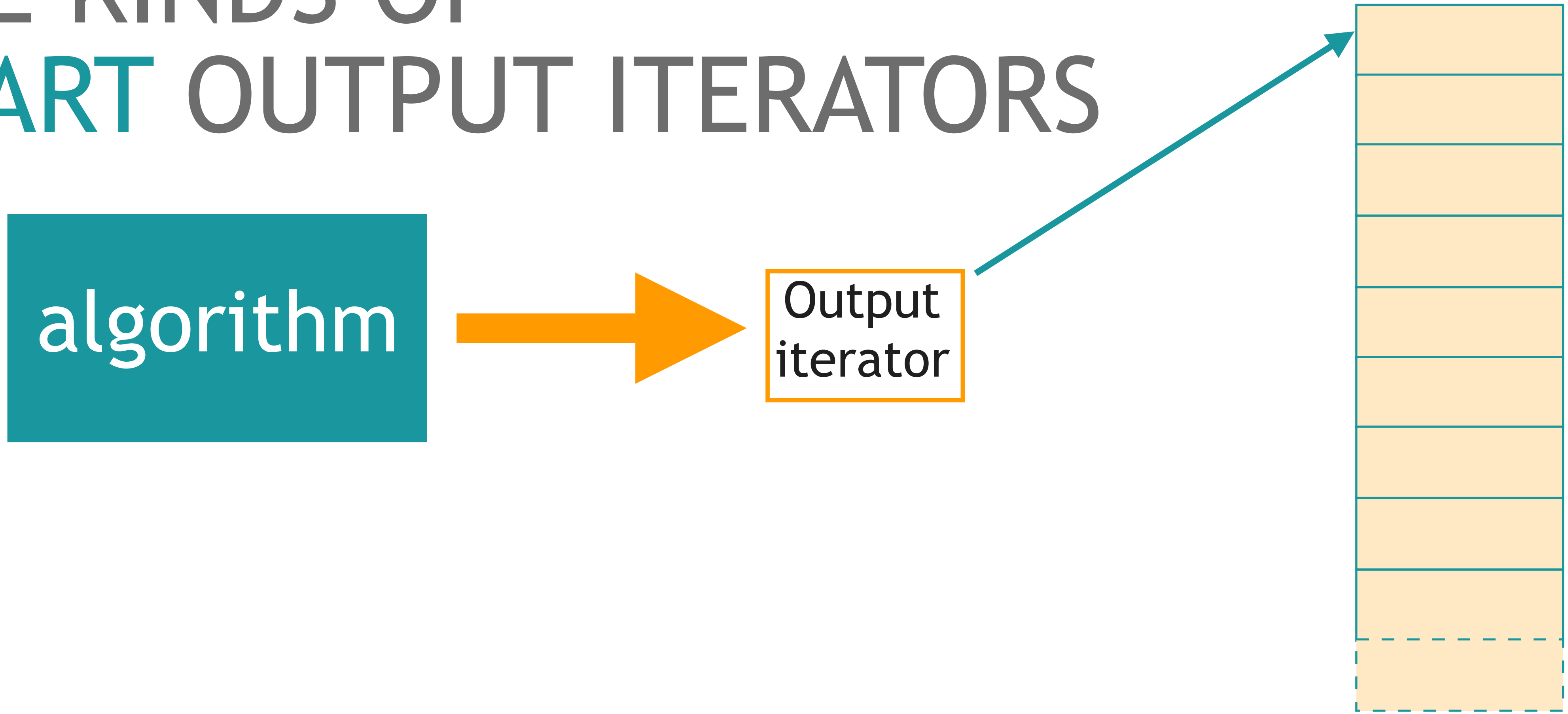
STL

SMART OUTPUT ITERATORS

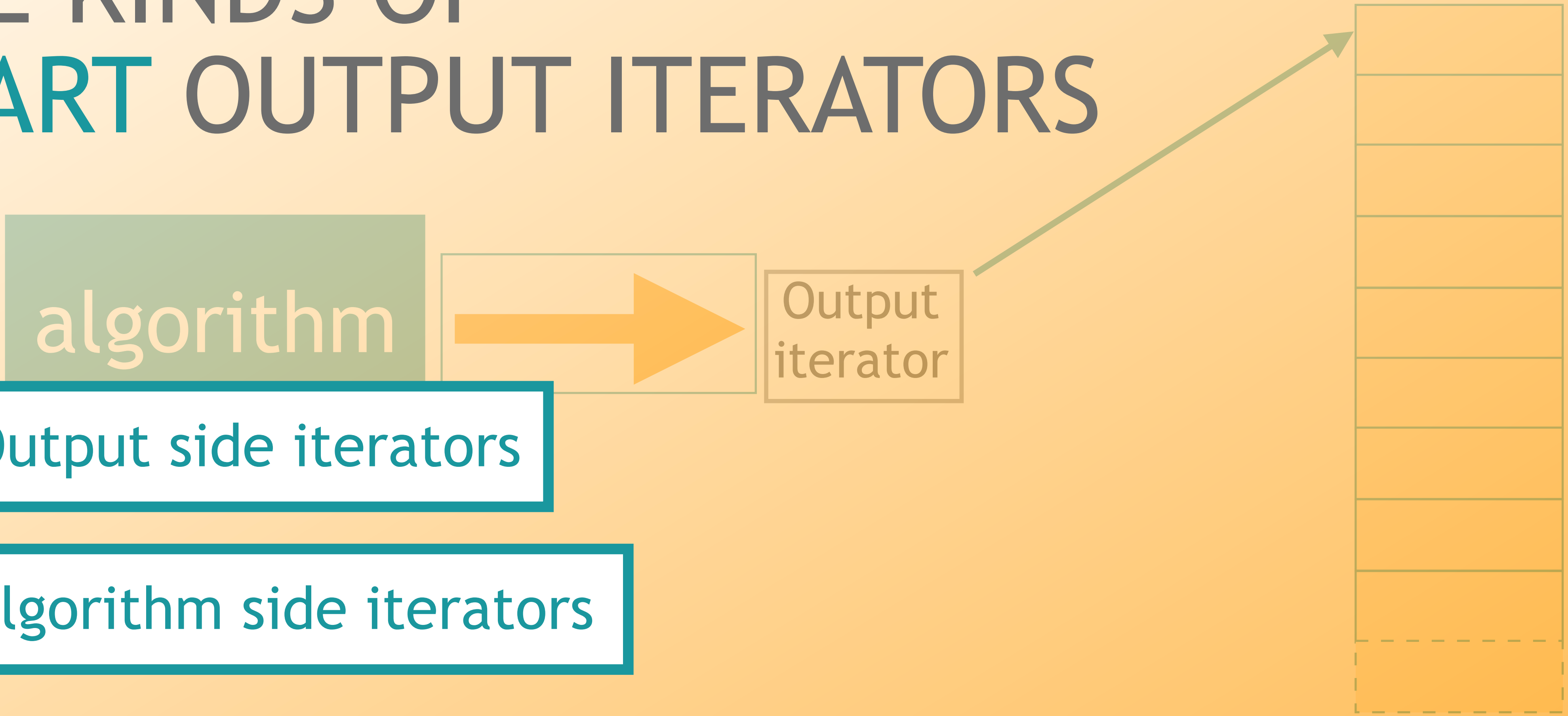


Dumb
Dumb
Input
iterators
algorithm
Output
iterator
Smart

PART 1 - THE KINDS OF SMART OUTPUT ITERATORS



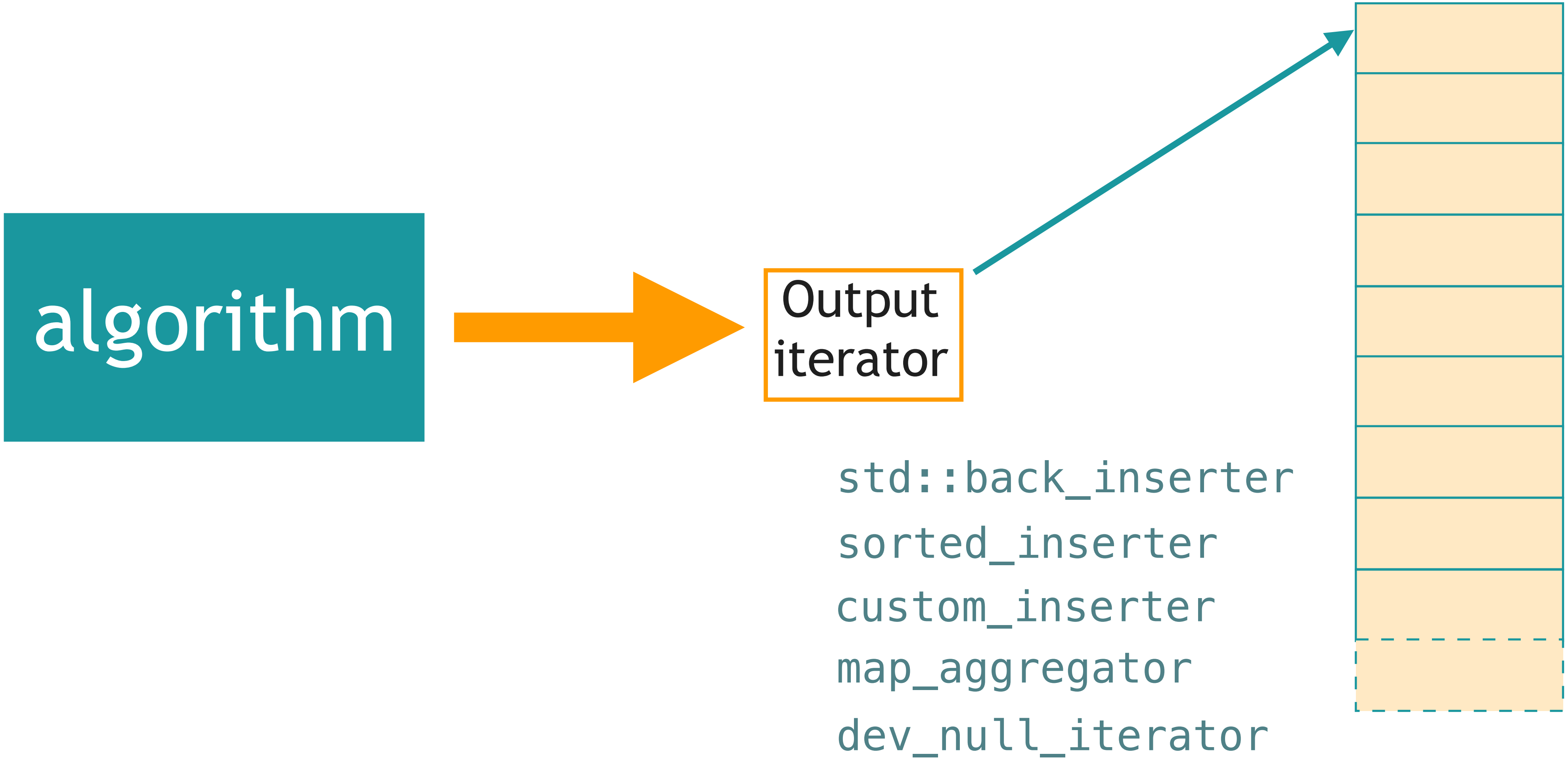
PART 1 -
THE KINDS OF
SMART OUTPUT ITERATORS



1a. Output side iterators

1b. Algorithm side iterators

TWO SIDES


algorithm
Output iterator
std::back_inserter
sorted_inserter
custom_inserter
map_aggregator
dev_null_iterator

WHAT GOES INTO AN OUTPUT ITERATOR?

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

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

`results` is `{1, 2, 3, 4, 5}`

MUREX™ Financial Software

Fluent{C++}

WHAT GOES INTO AN OUTPUT ITERATOR?

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

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

```
results is {0, 0, 0, 0, 0, 1, 2, 3, 4, 5}
```

std::copy

```
for (auto current = first; current != last; ++current)
{
    *out = *current;
    ++out;
    ++current;
}
```

BACK_INSERTER

```
std::copy
for (auto current = first; current != last; ++current)
{
    *out = *current;
    ++out;
    ++current;
}
```

```
class back_insert_iterator
{
public:
    explicit back_insert_iterator(Container& container) : container_(container) {}

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

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

    back_insert_iterator& operator=(typename Container::value_type const& value)
    {
        container_.push_back(value);
        return *this;
    }

private:
    Container& container_;
};
```

INSERTER

```
std::vector<int> numbers = {1, 3, -1, 0, 1, 3, 0};
std::set<int> results;

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

results is {-1, 0, 1, 3}

MUREX™ Financial Software

Fluent{C++}

INSERTER

```
std::vector<int> numbers = {1, 3, -1, 0, 1, 3, 0};
std::set<int> results;

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

results is `{-1, 0, 1, 3}`

MUREX™ Financial Software

Fluent{C++}

SORTED_INSERTER

```
class sorted_insert_iterator
{
public:
    explicit sorted_insert_iterator(Container& container) : container_(container) {}

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

    sorted_insert_iterator& operator*() { return *this; }

    sorted_insert_iterator& operator=(typename Container::value_type const& value)
    {
        container_.insert(value);
        return *this;
    }

private:
    Container& container_;
};
```

+ hint

CUSTOM_INSERTER

```
void legacyInsert(int number, DarkLegacyStructure& thing);

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

DarkLegacyStructure legacyStructure = // ...

std::copy(begin(input), end(input),
   custom_inserter([&legacyStructure](int number){ legacyInsert(number, legacyStructure); });
```

CUSTOM_INSERTER

```
template<typename InsertFunction>
class custom_insert_iterator
{
public:
    explicit custom_insert_iterator(InsertFunction 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:
    InsertFunction insertFunction_;
};
```

MAP_AGGREGATOR

```
std::map<int, std::string> entries = { {1, "a"}, {2, "b1"}, {3, "c1"} };
std::map<int, std::string> entries2 = { {2, "b2"}, {3, "c2"}, {4, "d"} };

std::map<int, std::string> results;

std::copy(begin(entries), end(entries), sorted_inserter(results));

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

results is `{ {1, "a"}, {2, "b1"}, {3, "c1"} }`

results is `{ {1, "a"}, {2, "b1b2"}, {3, "c1c2"}, {4, "d"} }`

MAP_AGGREGATOR

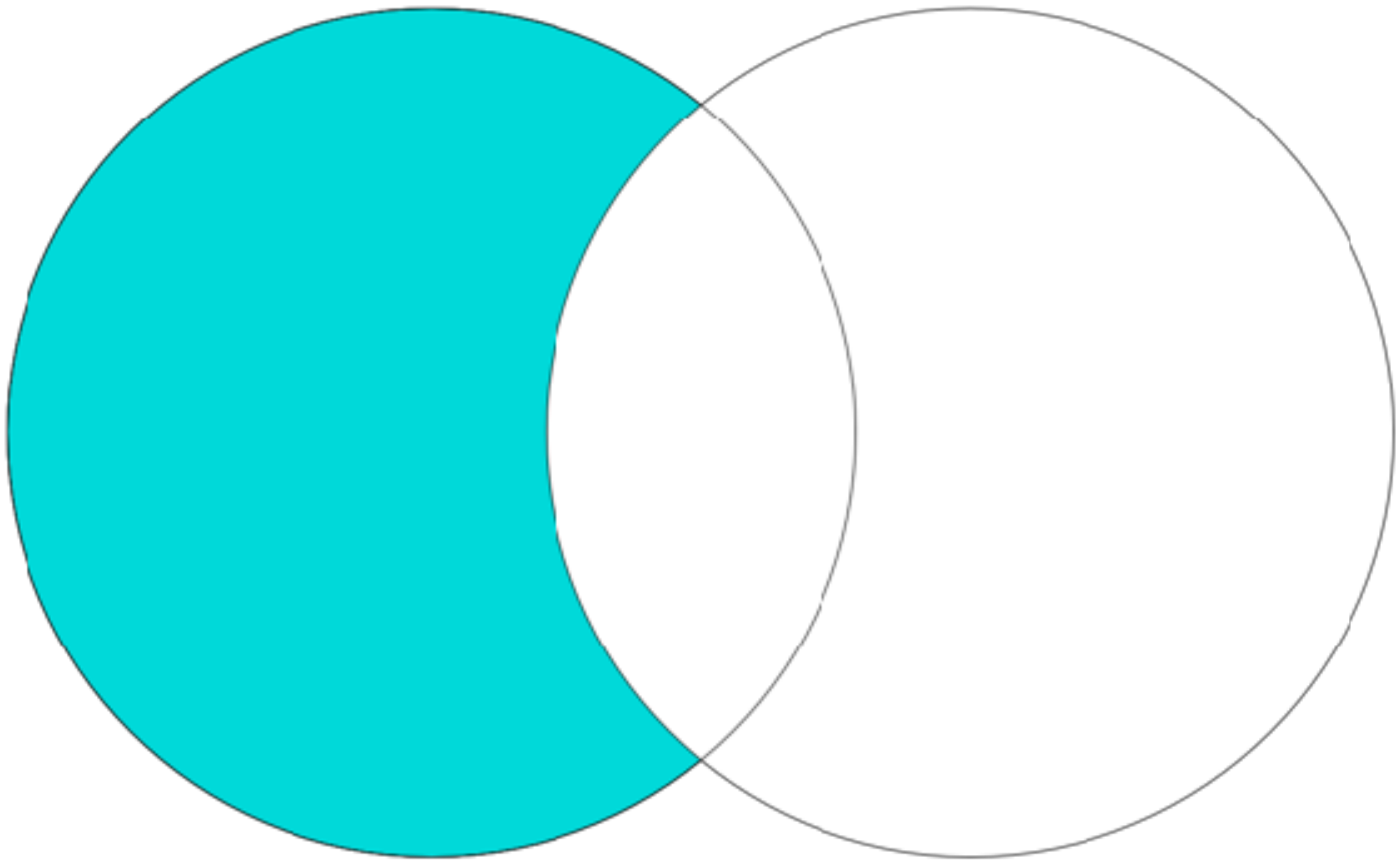
```
template<typename Map, typename Function>
class map_aggregate_iterator
{
public:
    map_aggregate_iterator(Map& map, Function aggregator) : map_(map), aggregator_(aggregator) {}

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

    map_aggregate_iterator& operator=(typename Map::value_type const& keyValue)
    {
        auto position = map_.find(keyValue.first);
        if (position != map_.end())
        {
            position->second = aggregator_(position->second, keyValue.second);
        }
        else
        {
            map_.insert(keyValue);
        }
        return *this;
    }

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

DEV_NULL_ITERATOR

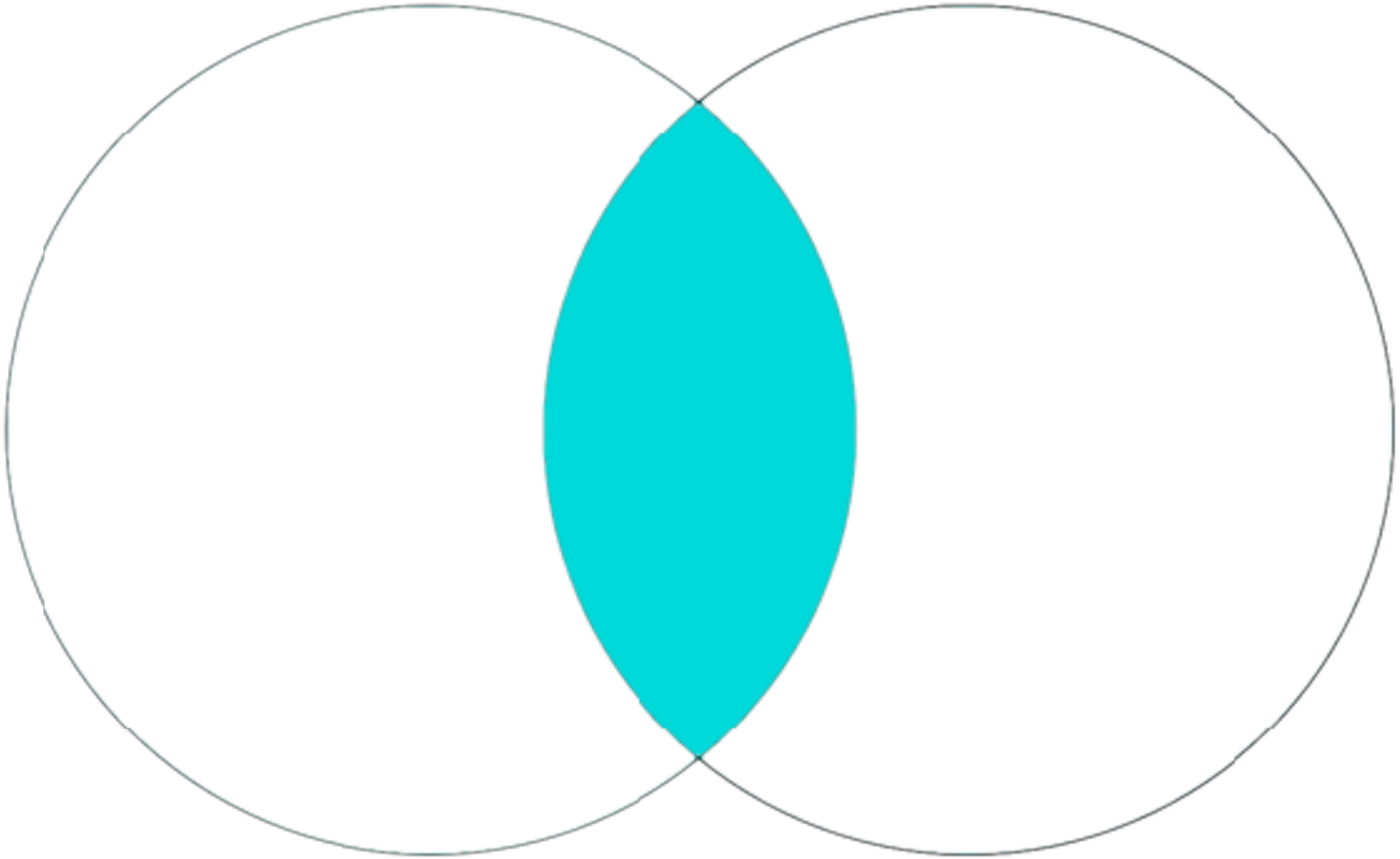
```
class dev_null_iterator
{
public:
    dev_null_iterator& operator++() { return *this; }

    dev_null_iterator& operator*() { return *this; }

    template<typename T>
    dev_null_iterator& operator=(T const&) { return *this; }
};
```

```
std::vector<int> numbers = {1, 3, -1, 0, 1, 3, 0};

std::copy(begin(numbers), end(numbers), dev_null_iterator());
```

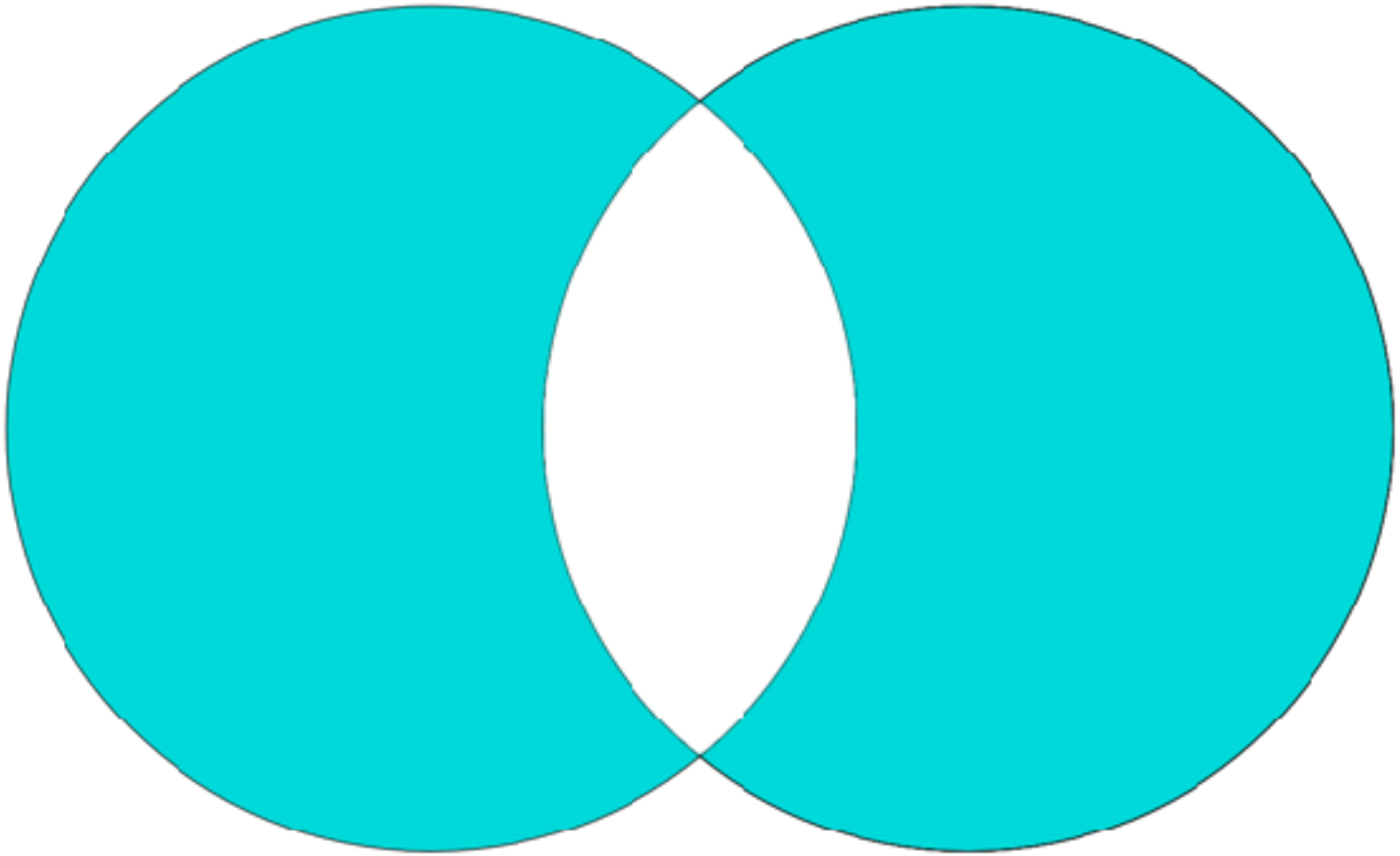
SET_DIFFERENCE



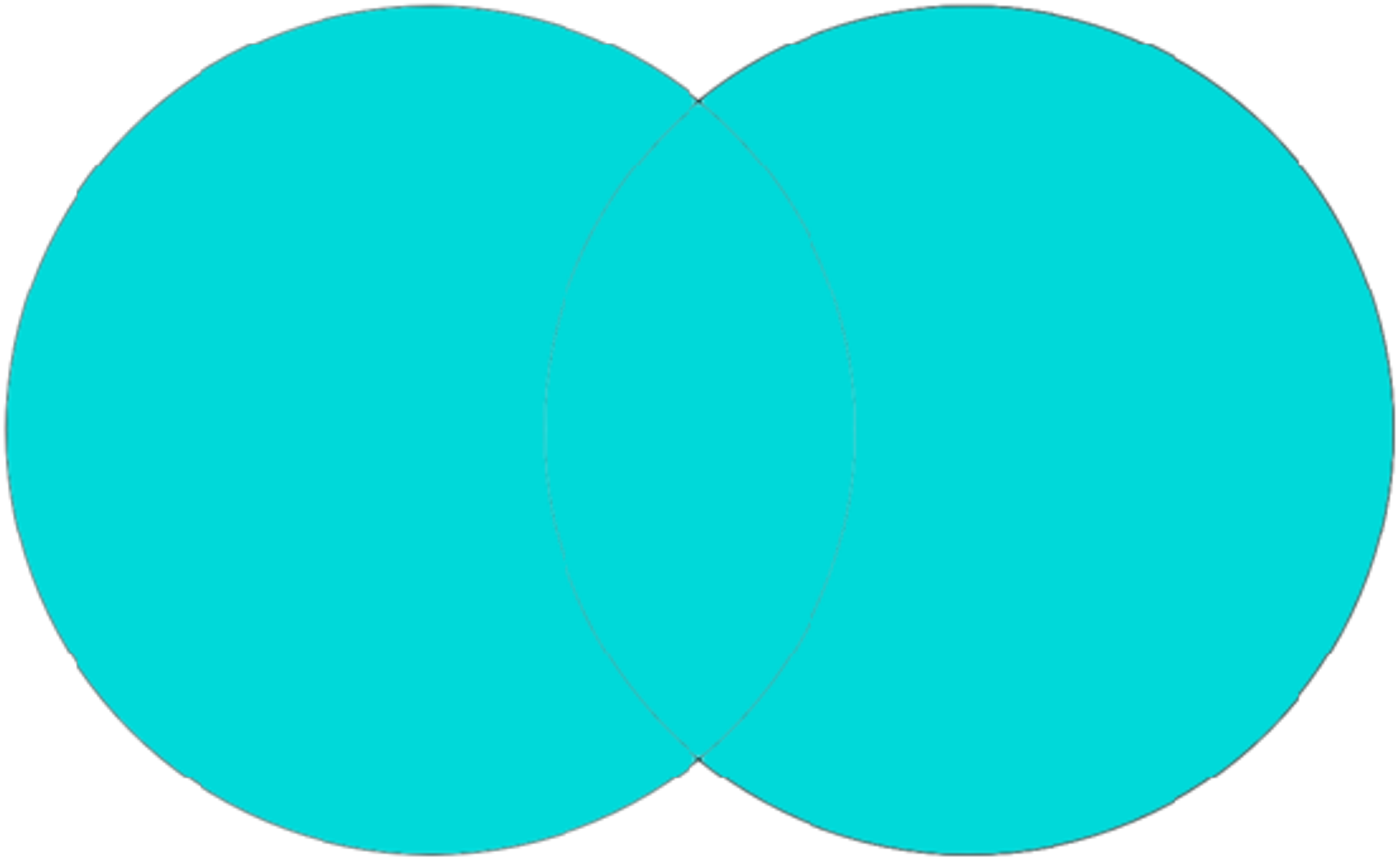
SET_INTERSECTION

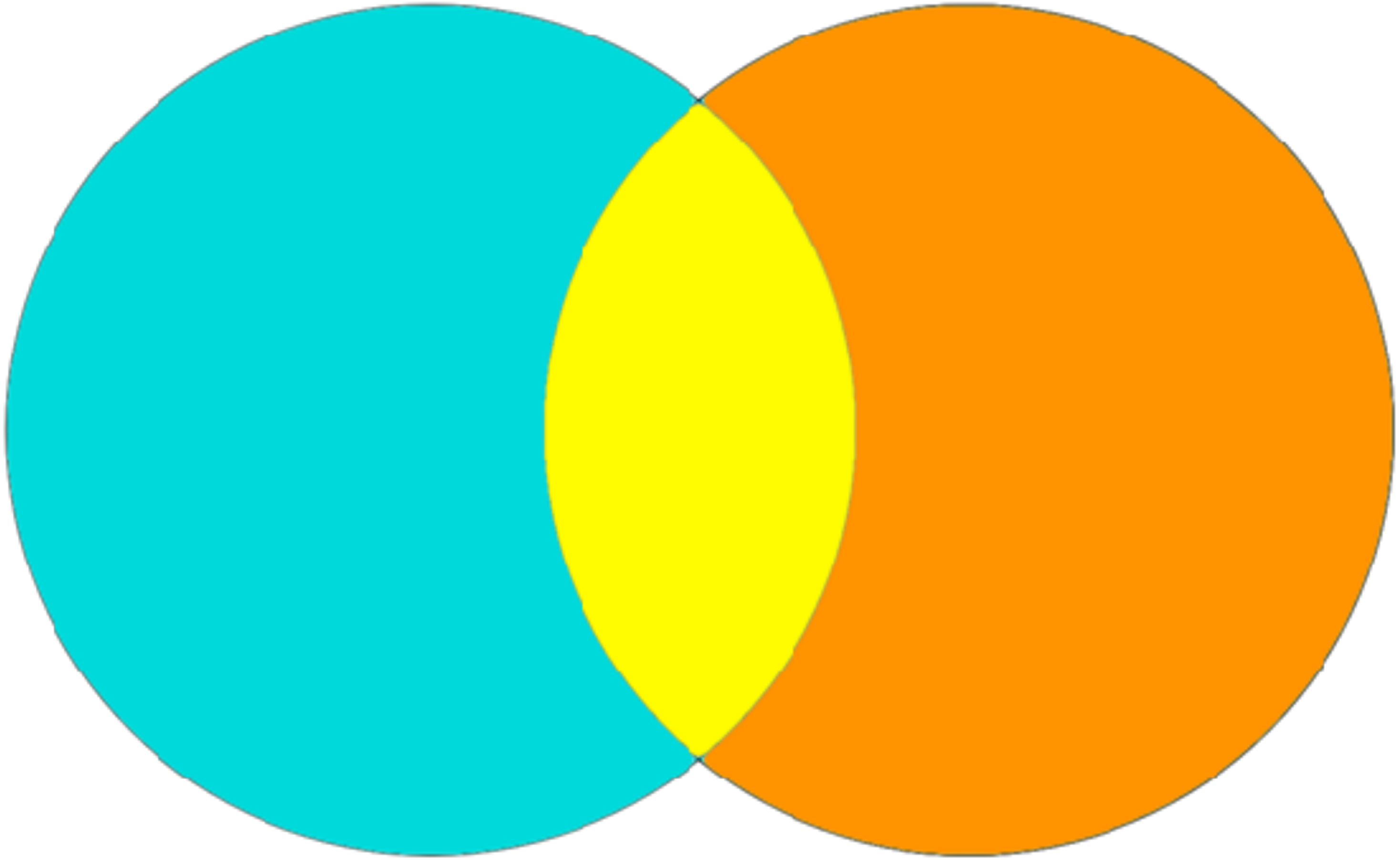


SET_SYMMETRIC_DIFFERENCE

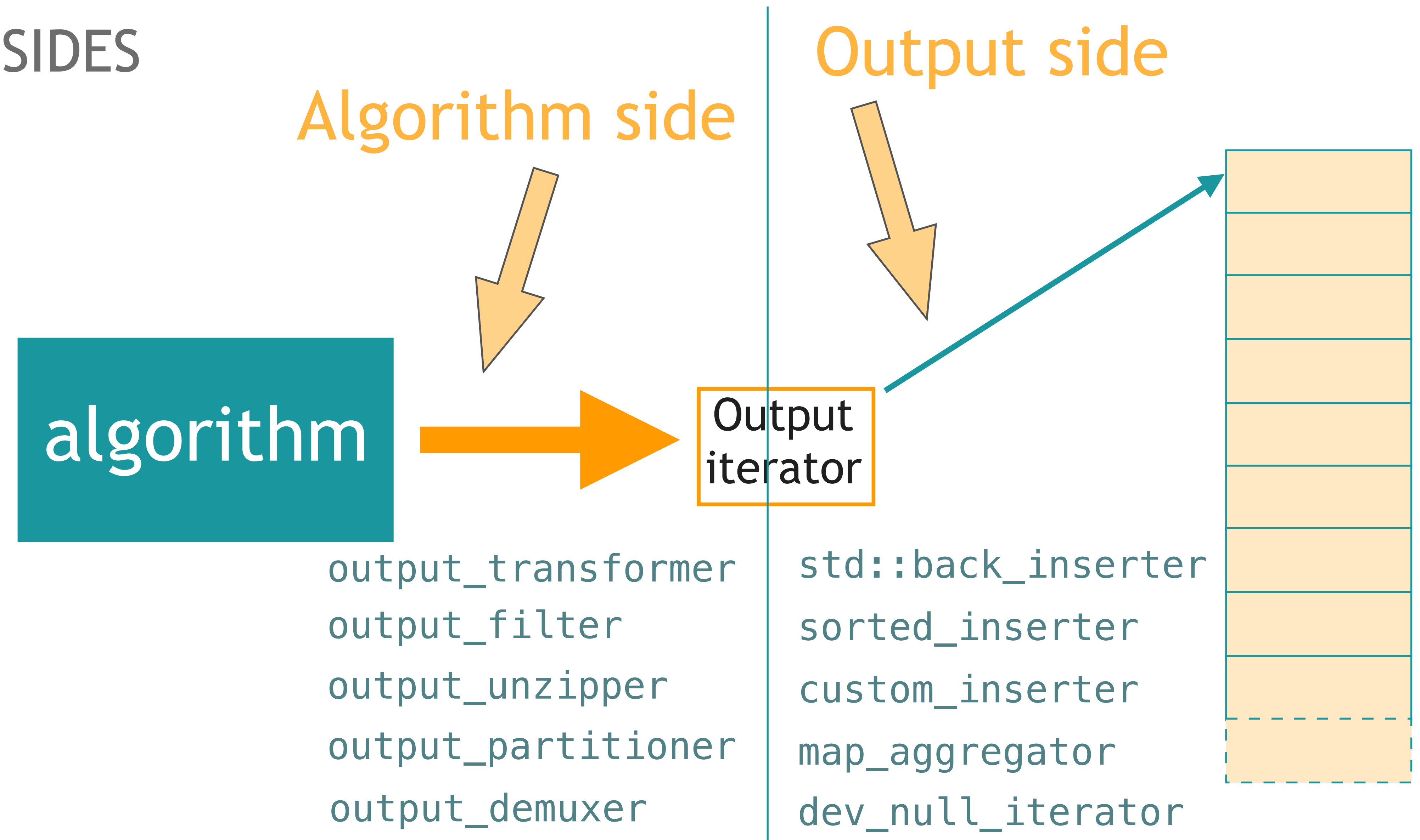


SET_UNION

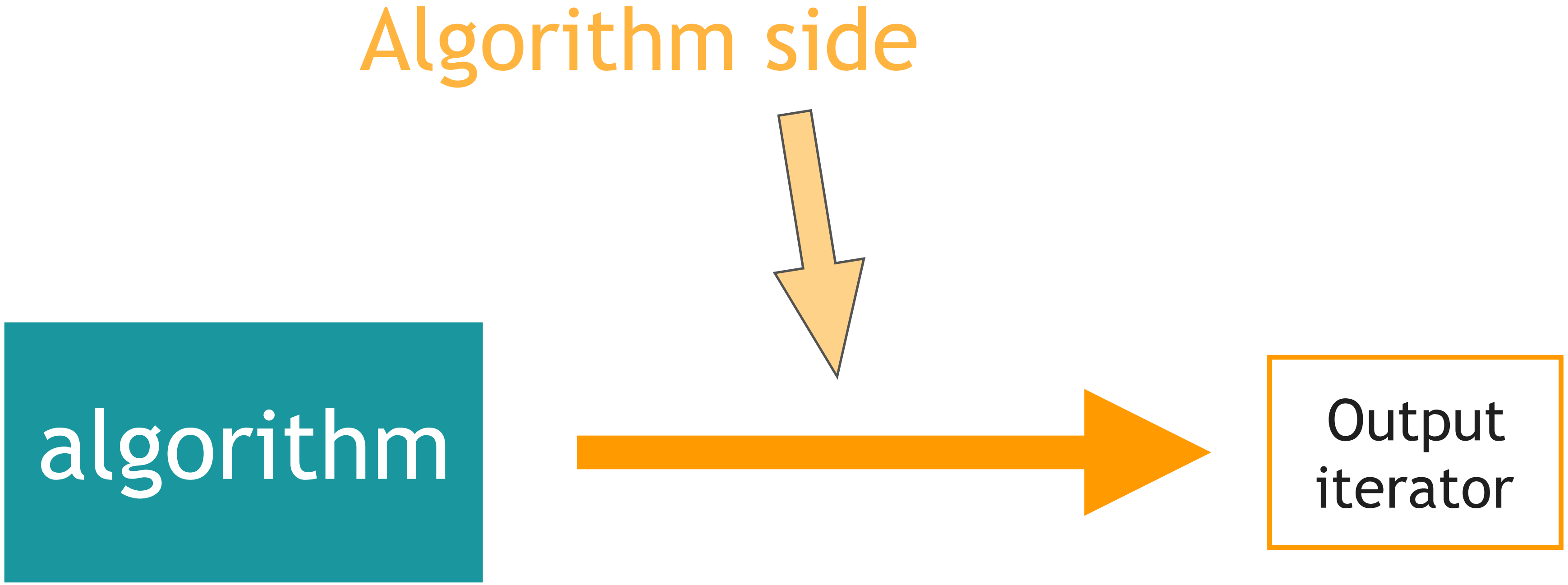




TWO SIDES


Algorithm side
Output side
algorithm
Output iterator
output_transformer
output_filter
output_unzipper
output_partitioner
output_demuxer
std::back_inserter
sorted_inserter
custom_inserter
map_aggregator
dev_null_iterator

TWO SIDES




```
output_transformer
output_filter
output_unzipper
output_partitioner
output_demuxer
```

OUTPUT_TRANSFORMER

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

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

std::vector<int> results;
std::copy(begin(input), end(input), times2(std::back_inserter(results)));
```

results is {2, 4, 6, 8, 10}

OUTPUT_TRANSFORMER

```
template<typename Iterator, typename TransformFunction>
class output_transform_iterator
{
public:
    explicit output_transform_iterator(Iterator iterator, TransformFunction transformFunction)
                : iterator_(iterator), transformFunction_(transformFunction) {}

    output_transform_iterator& operator++(){ ++iterator_; 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_;
};
```

OUTPUT_TRANSFORMER

```
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);
}
```

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

OUTPUT_FILTER

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

auto const ifIsEven = make_output_filter([](int i){ return i % 2 == 0; });

std::vector<int> results;
std::copy(begin(input), end(input), ifIsEven(std::back_inserter(results)));
```

results is {2, 4, 6, 8, 10}

OUTPUT_FILTER

```
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_;
};
```

More on this later

OUTPUT_FILTER

```
template<typename Predicate>
class output_filter
{
public:
    explicit output_filter(Predicate predicate) : predicate_(predicate) {}

    template<typename Iterator>
    output_filter_iterator<Iterator, Predicate> operator()(Iterator iterator) const
    {
        return output_filter_iterator<Iterator, Predicate>(iterator, predicate_);
    }

private:
    Predicate predicate_;
};
```

```
template<typename Predicate>
output_filter<Predicate> make_output_filter(Predicate predicate)
{
    return output_filter<Predicate>(predicate);
}
```

OUTPUT_PARTITIONER

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

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

std::vector<int> evens;
std::vector<int> odds;

std::copy(begin(input), end(input),
          isEvenPartition(std::back_inserter(evens), std::back_inserter(odds)));
```

```
evens is  {2, 4, 6, 8, 10}
 odds is  {1, 3, 5, 7, 9}
```

MUREX™ Financial Software

Fluent{C++}

OUTPUT_PARTITIONER

```
template<typename IteratorTrue, typename IteratorFalse, typename Predicate>
class output_partition_iterator
{
public:
    explicit output_partition_iterator(IteratorTrue iteratorTrue, IteratorFalse iteratorFalse, Predicate predicate)
            : iteratorTrue_(iteratorTrue), iteratorFalse_(iteratorFalse), predicate_(predicate) {}

    output_partition_iterator& operator++(){ ++iteratorTrue_; ++iteratorFalse_; return *this; }
    output_partition_iterator& operator*(){ return *this; }

    template<typename T>
    output_partition_iterator& operator=(T const& value)
    {
        if (predicate_(value))
        {
            *iteratorTrue_ = value;
        }
        else
        {
            *iteratorFalse_ = value;
        }
        return *this;
    }
private:
    IteratorTrue iteratorTrue_;
    IteratorFalse iteratorFalse_;
    Predicate predicate_;
};
```

MUREX™ Financial Software

Fluent{C++}

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_;
};
```

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

OUTPUT_UNZIPPER

```
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), output_unzipper(back_inserter(keys), back_inserter(values)));
```

keys is {1, 2, 3, 4, 5}

values is {"one", "two", "three", "four", "five"}

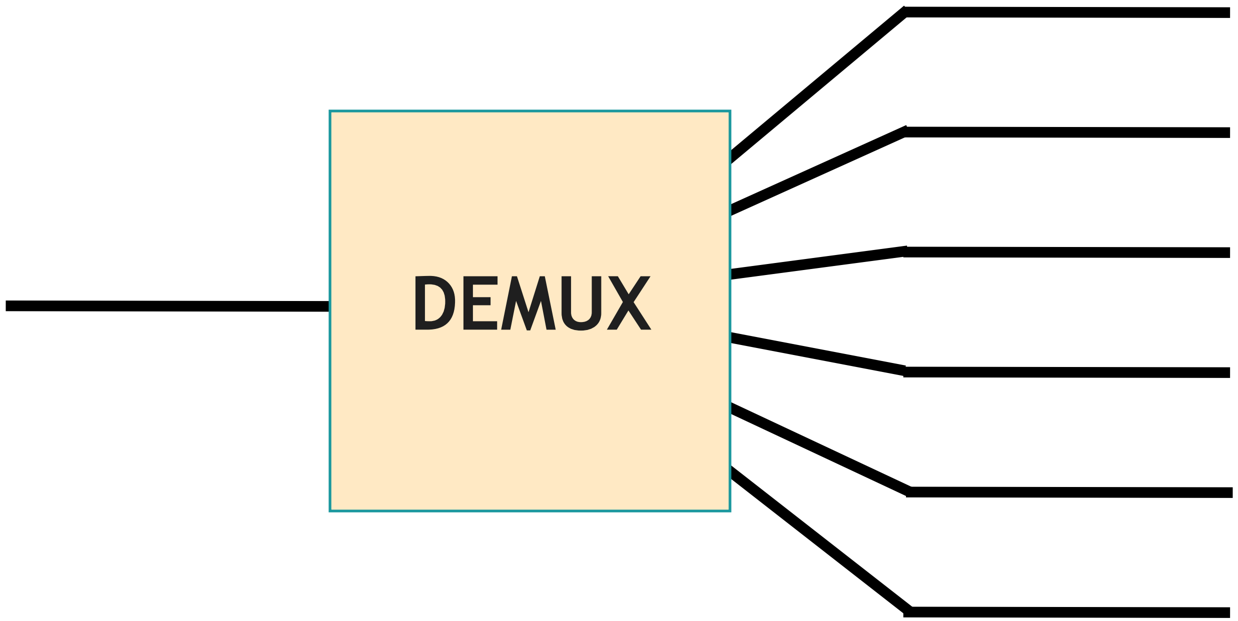
OUTPUT_UNZIPPER

```
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), output_unzipper(back_inserter(column1), back_inserter(column2), back_inserter(column3)));
```

column1 is {1, 4, 7, 10}
column2 is {2, 5, 8, 11}
column3 is {3, 6, 9, 12}

OUTPUT_UNZIPPER

```
template<typename... Iterators>
class output_unzip_iterator
{
public:
    explicit output_unzip_iterator(Iterators... iterators) : iterators_(std::make_tuple(iterators...)) {}
    output_unzip_iterator& operator++()
    {
        detail::apply([](auto&& iterator){ ++iterator; }, iterators_);
        return *this;
    }
    output_unzip_iterator& operator*(){ return *this; }

    template<typename... Ts>
    output_unzip_iterator& operator=(std::tuple<Ts...> const& values)
    {
        detail::apply2([](auto&& value, auto&& iterator){ *iterator = value; }, values, iterators_);
        return *this;
    }
    template<typename First, typename Second>
    output_unzip_iterator& operator=(std::pair<First, Second> const& values)
    {
        *std::get<0>(iterators_) = values.first;
        *std::get<1>(iterators_) = values.second;
        return *this;
    }
private:
    std::tuple<Iterators...> iterators_;
};
```


DEMUX

OUTPUT_DEMUXER

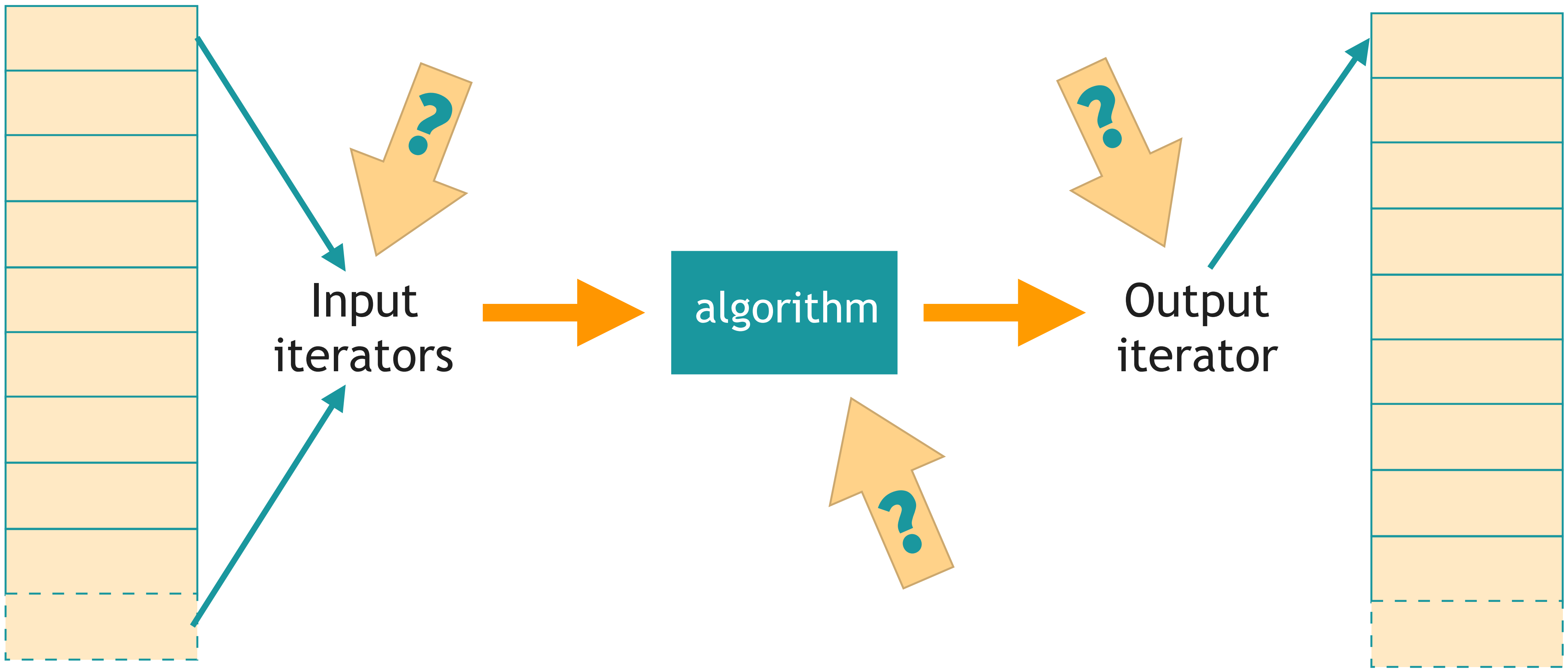
```
std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::vector<int> multiplesOf3;
std::vector<int> multiplesOf2Only;
std::vector<int> multiplesOf1Only;

std::copy(begin(numbers), end(numbers),
    output_demuxer(demux_if( [](int n){ return n % 3 == 0; } ).sendTo(back_inserter(multiplesOf3)),
                   demux_if( [](int n){ return n % 2 == 0; } ).sendTo(back_inserter(multiplesOf2Only)),
                   demux_if( [](int n){ return n % 1 == 0; } ).sendTo(back_inserter(multiplesOf1Only))));
```

```
    multiplesOf3 is {3, 6, 9}
multiplesOf2Only is {2, 4, 8, 10}
multiplesOf1Only is {1, 5, 7}
```

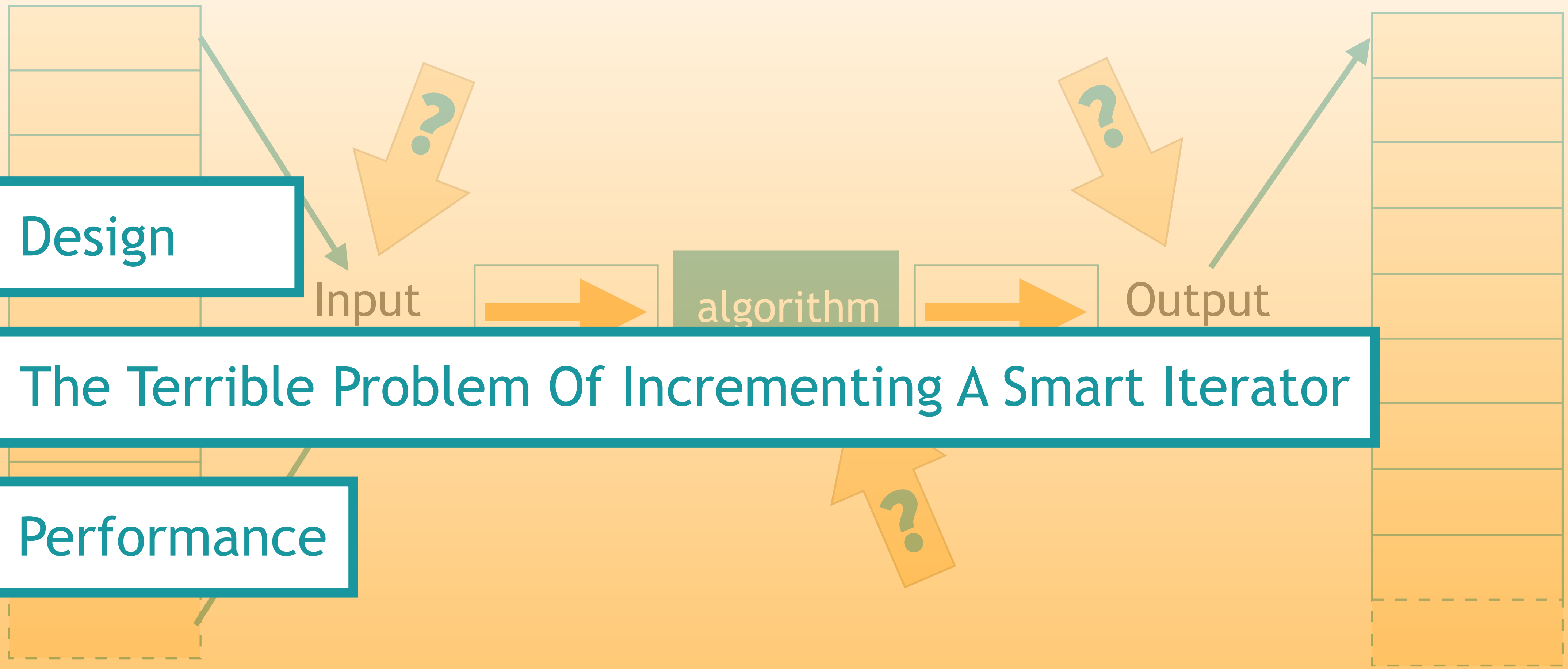
OUTPUT_DEMUXER

```
namespace detail
{
    template<typename Predicate>
    using ExecuteOnFirst_Predicate = detail::NamedType<Predicate, struct ExecuteOnFirst_Predicate_Tag>;

    template<typename Function>
    using ExecuteOnFirst_Function = detail::NamedType<Function, struct ExecuteOnFirst_Function_Tag>;

    template <typename Predicate, typename Function>
    struct Executor_on_first_that_satisfies_predicate
    {
        Executor_on_first_that_satisfies_predicate(ExecuteOnFirst_Predicate<Predicate> p, ExecuteOnFirst_Function<Function>
f) : p_(p.get()), f_(f.get()) {}
        Predicate p_;
        Function f_;
        bool hasExecutedAlready = false;

        template<typename T>
        void operator()(T&& value)
        {
            if (!hasExecutedAlready)
            {
                if (p_(value))
                {
                    f_(value);
                    hasExecutedAlready = true;
                }
            }
        }
    };

    template<typename Predicate, typename Function>
    Executor_on_first_that_satisfies_predicate<Predicate, Function>
make_executor_on_first_that_satisfies_predicate(ExecuteOnFirst_Predicate<Predicate> p, ExecuteOnFirst_Function<Function> f)
    {
        return Executor_on_first_that_satisfies_predicate<Predicate, Function>(p, f);
    }

    template <typename Tuple, typename Predicate, typename Function>
    void execute_on_first_that_satisfies_predicate(Tuple&& tuple, ExecuteOnFirst_Predicate<Predicate> p,
ExecuteOnFirst_Function<Function> f)
    {
        auto executor_on_first_that_satisfies_predicate = detail::make_executor_on_first_that_satisfies_predicate(p, f);
        apply(executor_on_first_that_satisfies_predicate, tuple);
    }

} // namespace detail

template<typename Predicate, typename Iterator>
struct demux_branch
{
    using iterator_type = Iterator;
    Predicate predicate;
    Iterator iterator;
    demux_branch(Predicate predicate, Iterator iterator) : predicate(predicate), iterator(iterator) {}
};

template<typename... DemuxBranches>
class output_demux_iterator
{
public:
    using iterator_category = std::output_iterator_tag;
    using value_type = void;
    using difference_type = void;
    using pointer = void;
    using reference = void;

    explicit output_demux_iterator(DemuxBranches const&... demuxBranches) :
branches_(std::make_tuple(demuxBranches...)) {}
    output_demux_iterator& operator++() { return *this; }
    output_demux_iterator& operator++(int){ ++*this; return *this; }
    output_demux_iterator& operator*(){ return *this; }
    template<typename T>
    output_demux_iterator& operator=(T&& value)
    {
        execute_on_first_that_satisfies_predicate(branches_,

detail::make_named<detail::ExecuteOnFirst_Predicate>([&value](auto&& branch){ return
branch.predicate(value); }),
                                                  detail::make_named<detail::ExecuteOnFirst_Function>([&value]
(auto&& branch){ *branch.iterator = value; ++branch.iterator; } ));

        return *this;
    }

    output_demux_iterator& operator=(output_demux_iterator const&) = default;
    output_demux_iterator& operator=(output_demux_iterator&&) = default;
    output_demux_iterator(output_demux_iterator const&) = default;
    output_demux_iterator(output_demux_iterator&&) = default;

private:
    std::tuple<DemuxBranches...> branches_;
};

template<typename... DemuxBranches>
output_demux_iterator<DemuxBranches...> output_demuxer(DemuxBranches const&... demuxBranches)
{
    return output_demux_iterator<DemuxBranches...>(demuxBranches...);
}

template<typename Predicate>
class Demux_if
{
public:
    Demux_if(Predicate predicate) : predicate_(std::move(predicate)) {}

    template<typename Iterator>
    auto sendTo(Iterator&& iterator) const &
    {
        return demux_branch<Predicate, Iterator>(predicate_, std::forward<Iterator>(iterator));
    }

    template<typename Iterator>
    auto sendTo(Iterator&& iterator) &&
    {
        return demux_branch<Predicate, Iterator>(std::move(predicate_), std::forward<Iterator>(iterator));
    }

private:
    Predicate predicate_;
};

template<typename Predicate>
Demux_if<Predicate> demux_if(Predicate&& predicate)
{
    return Demux_if<Predicate>(std::forward<Predicate>(predicate));
}
```

PART 2 -WHERE TO PUT THE LOGIC?



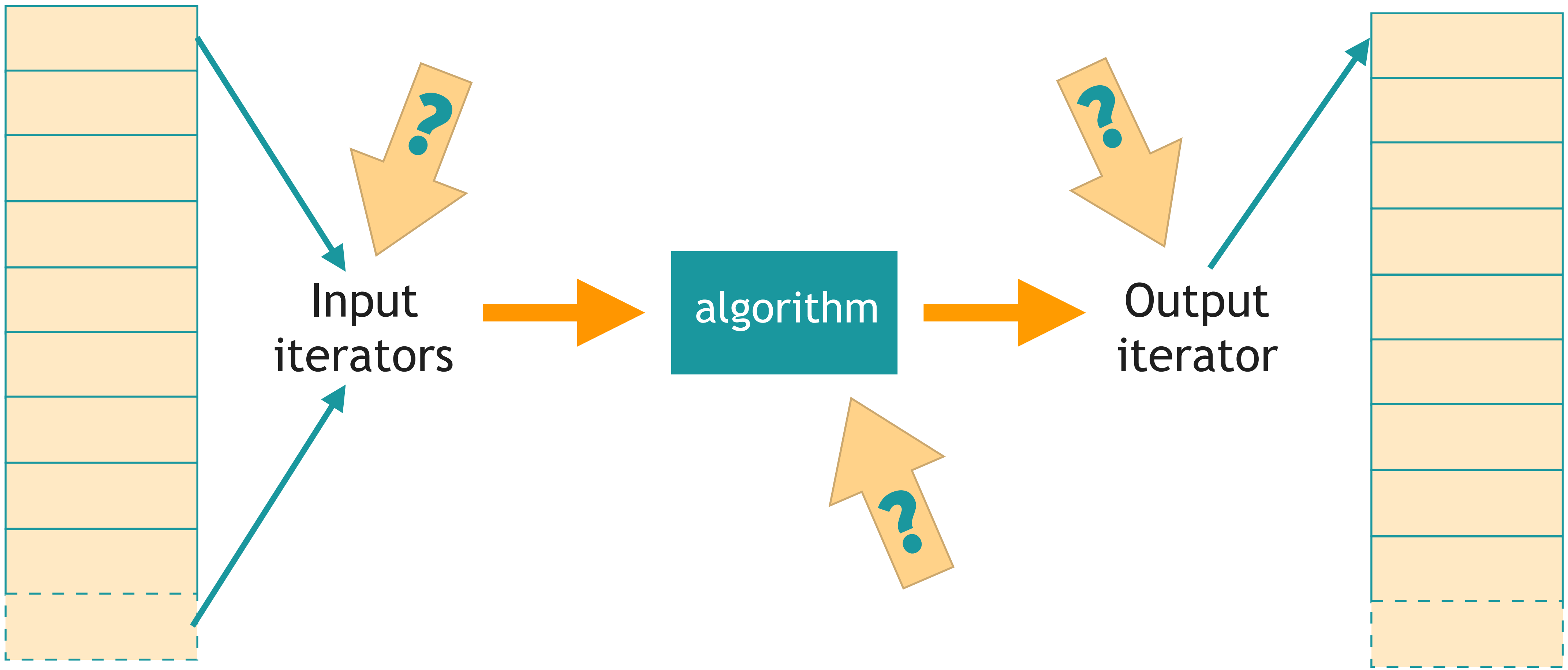
PART 2 -WHERE TO PUT THE LOGIC?



2a. Design

2b. The Terrible Problem Of Incrementing A Smart Iterator

2c. Performance


Input
iterators
algorithm
Output
iterator
?
?
?

WHERE TO PUT THE LOGIC?

algorithm

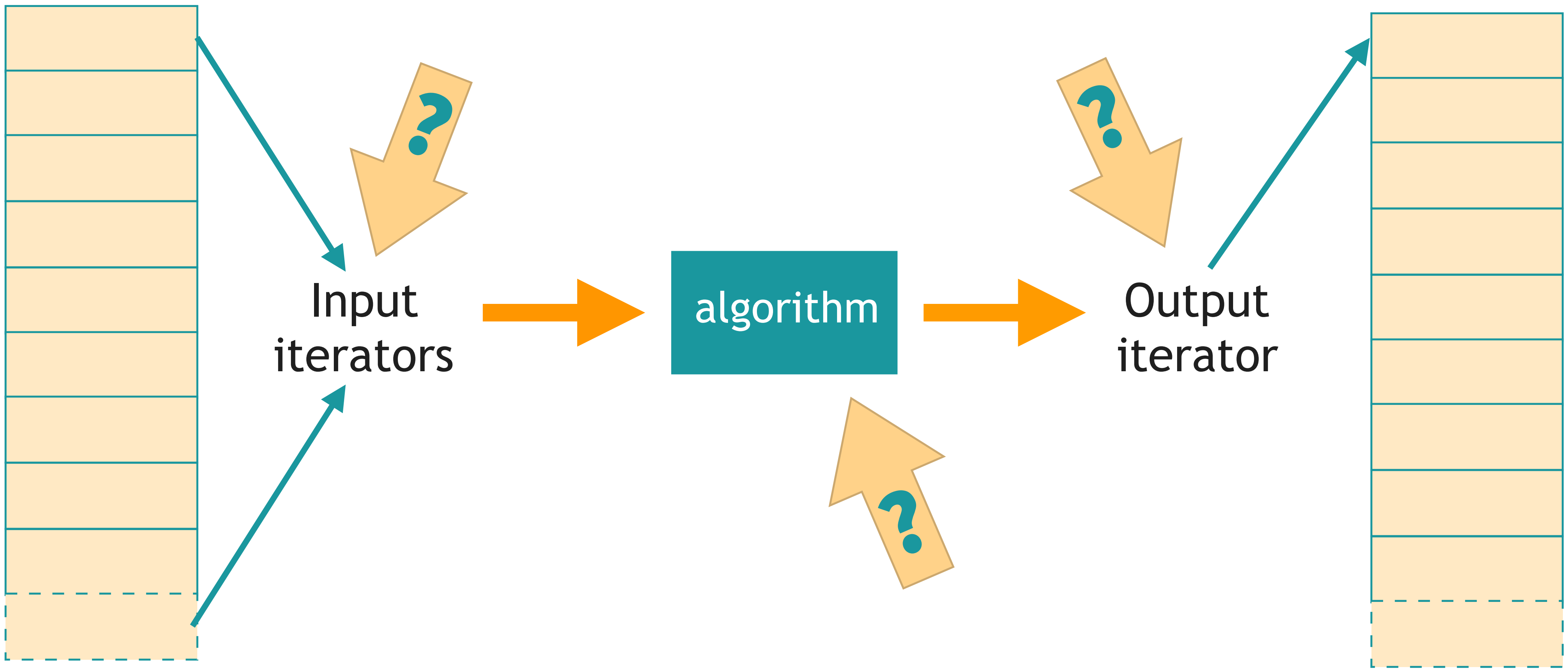
Advantages:

- Mostly standard
- Easily integrable if not standard
- Everyone (should) knows them by heart

Drawbacks:

- Not composable

WHERE TO PUT THE LOGIC?



WHERE TO PUT THE LOGIC?

Input iterators

Advantages:

- Composable
- Pipe syntax very expressive

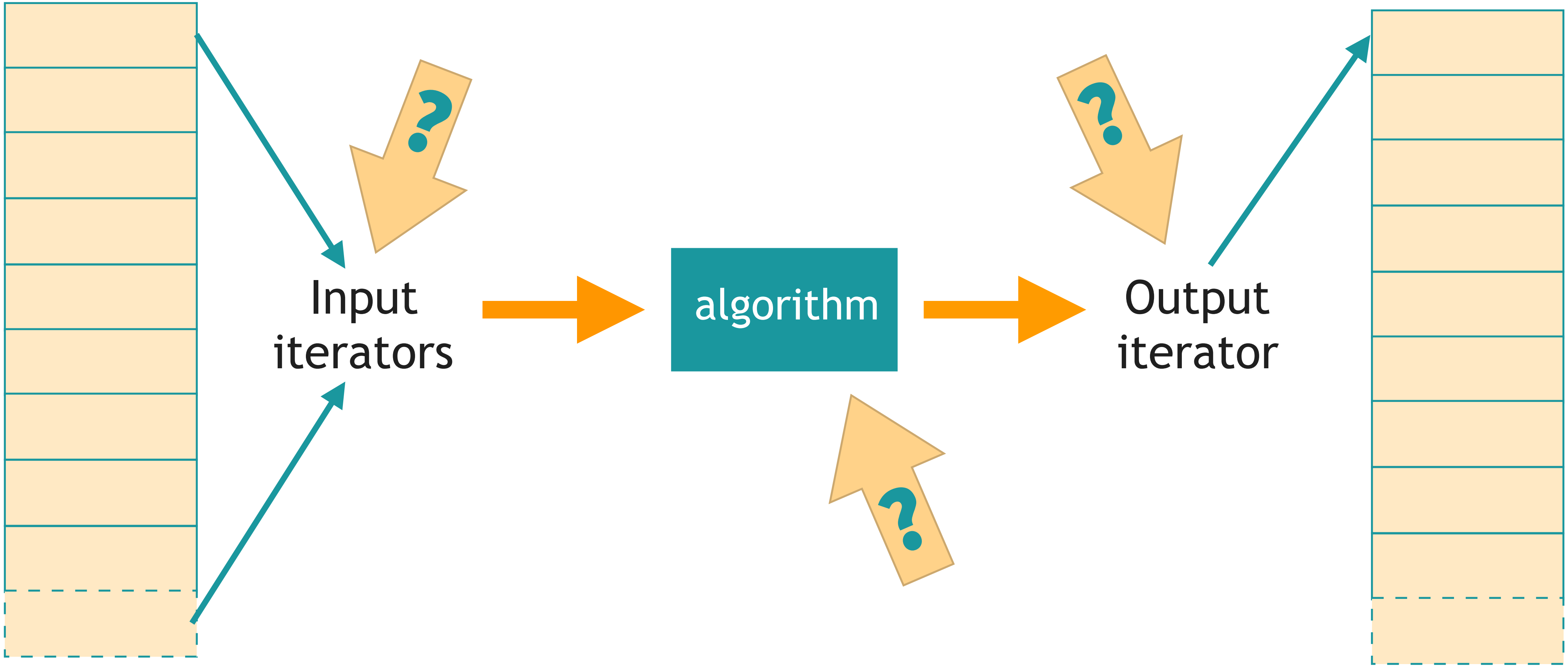
```
numbers | view::filter(isEven) | view::transform(times2)
```

- Work with multiple inputs algorithms

Drawbacks:

- Not standard (even in C++20?)
- Hard to implement/compile
- Don’t work with multiple outputs

WHERE TO PUT THE LOGIC?



WHERE TO PUT THE LOGIC?

Output iterator

Advantages:

- Works with multiple output algorithms
- Acts on the output container
- Composable, pipeable

```
ifIsEven(times2(std::back_inserter(results)))
ifIsEven | times2 | std::back_inserter(results)
```

- Easy to implement

Drawbacks:

- Not standard

THE TERRIBLE PROBLEM OF INCREMENTING A SMART ITERATOR

Fluent {C++}

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

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

```cpp
// 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 << ' ';
}
```

```
4 8
```

MUREX™
Financial Software

Fluent{C++}

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

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

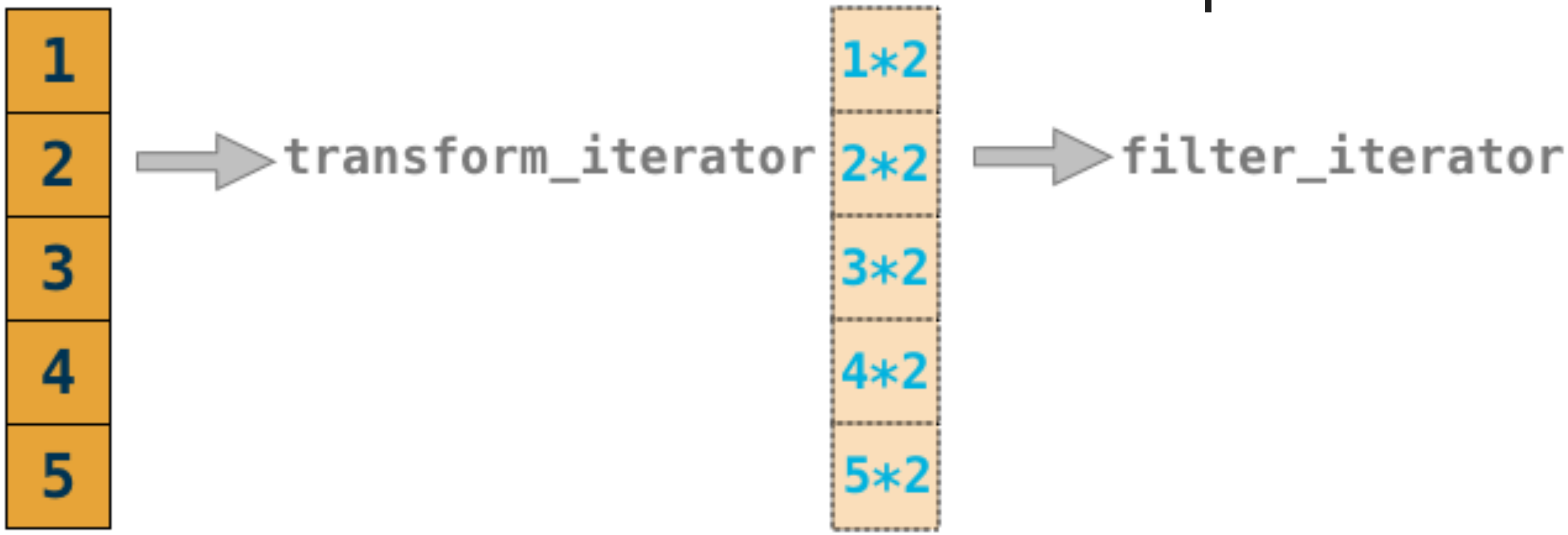
```cpp
// Input vector
std::vector<int> numbers = {1, 2, 3, 4, 5};

// 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 << ' ';
}
```

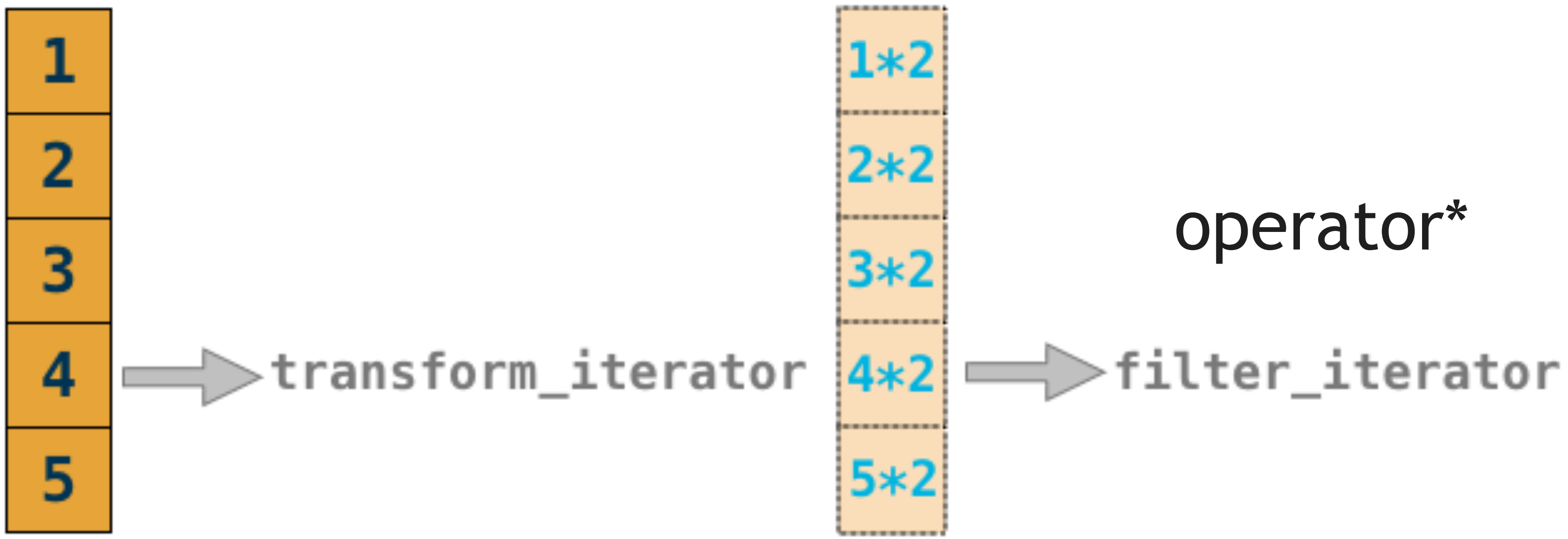
```
transform 1
transform 2
transform 2
transform 3
transform 4
transform 4
transform 5
4 8
```

MUREX™ Financial Software

Fluent{C++}

operator++




1
2
3
4
5
transform_iterator
1*2
2*2
3*2
4*2
5*2
operator*
filter_iterator

OUTPUT_FILTER

```
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_;
};
```

More on this later

MUREX™
FINANCIAL SOFTWARE

Fluent{C++}

```cpp
// 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 << ' ';
}
```

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

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

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

```cpp
// 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 << ' ';
}
```

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

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

```cpp
std::copy
for (auto current = first; current != last; ++current)
{
    *out = *current;
    ++out;
    ++current;
}
```

```
0 4 0 8 0
```


MUREX™
Financial Software

Fluent{C++}

OUTPUT_FILTER

```
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_;
};
```

OUTPUT_FILTER

```
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;
        }
        return *this;
    }

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

MUREX™ Financial Software

Fluent{C++}

```
// 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 << ' ';
}
```

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

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

```
std::copy
for (auto current = first; current != last; ++current)
{
    *out = *current;
    ++out;
    ++current;
}
```

```
8 0 0 0 0
```

MUREX™ Financial Software

Fluent{C++}

OUTPUT_FILTER

```
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;
        }
        return *this;
    }

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

OUTPUT_FILTER

```
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_;
};
```

Is this ok?

Assignment through an output iterator is expected to alternate with incrementing. Double-increment is undefined behavior.[1]

[1] cppreference.com

```cpp
// 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 << ' ';
}
```

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

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

```cpp
std::copy
for (auto current = first; current != last; ++current)
{
    *out = *current;
    ++out;
    ++current;
}
```

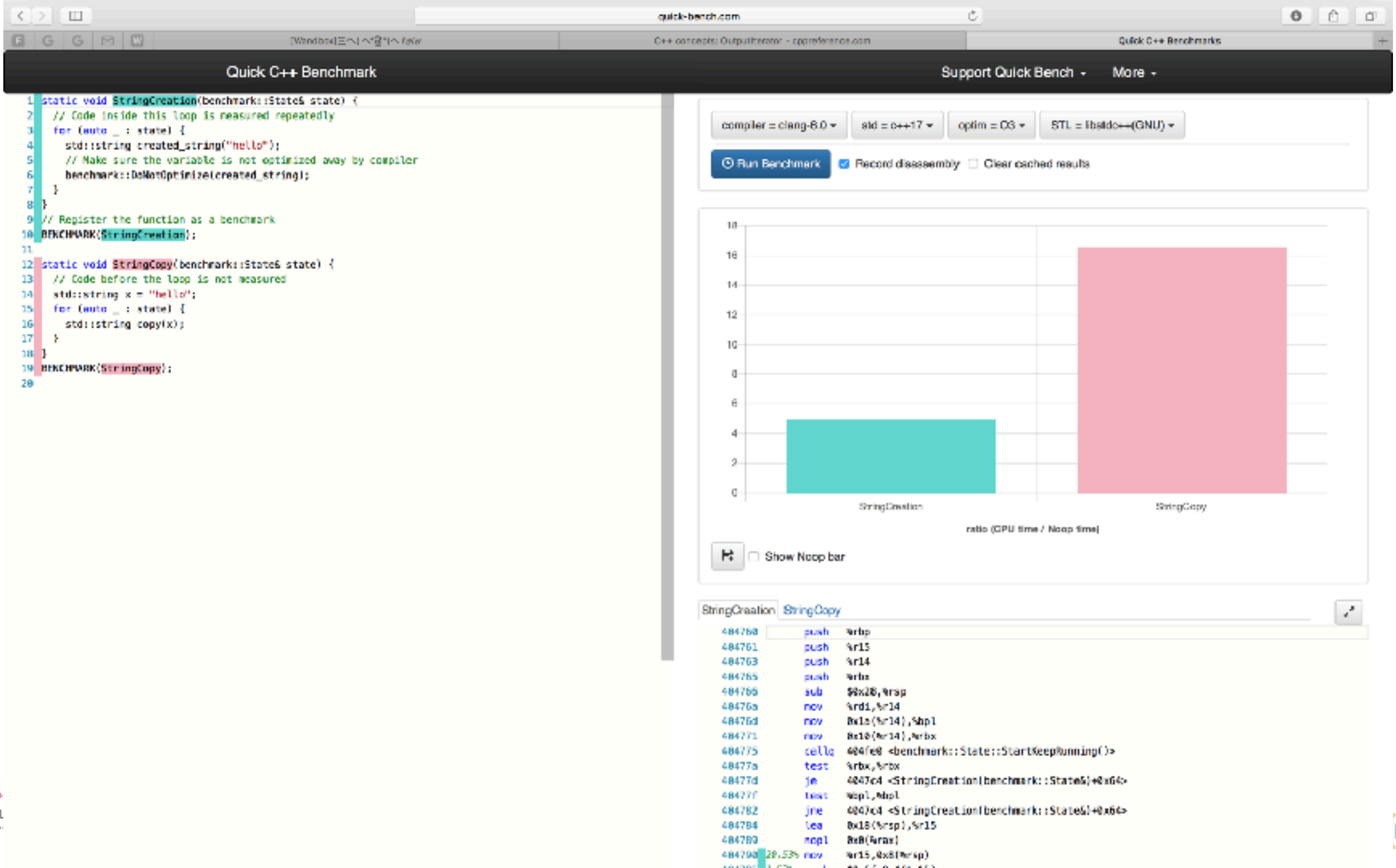
```
4 8 0 0 0
```

MUREX™
FINANCIAL SOFTWARE

Fluent{C++}

PERFORMANCE BENCHMARKS

quick-bench.com



quick-bench.com
Quick C++ Benchmark
Support Quick Bench
More
static void StringCreation(benchmark::State& state) {
// Code inside this loop is measured repeatedly
for (auto _ : state) {
std::string created_string("hello");
// Make sure the variable is not optimized away by compiler
benchmark::DoNotOptimize(created_string);
}
}
// Register the function as a benchmark
BENCHMARK(StringCreation);
static void StringCopy(benchmark::State& state) {
// Code before the loop is not measured
std::string x = "hello";
for (auto _ : state) {
std::string copy(x);
}
}
BENCHMARK(StringCopy);
compiler = clang-6.0
std = c++17
optim = O3
STL = libstdc++(GNU)
Run Benchmark
Record disassembly
Clear cached results
18
16
14
12
10
8
6
4
2
0
StringCreation
StringCopy
ratio (CPU time / Noop time)
Show Noop bar
StringCreation
StringCopy
404760 push %rbp
404761 push %r15
404763 push %r14
404765 push %rbx
404766 sub $0x28,%rsp
40476a mov %rdi,%r14
40476d mov 0x1a(%r14),%bpl
404771 mov 0x10(%r14),%rbx
404775 callq 404fe0 <benchmark::State::StartKeepRunning()>
40477a test %rbx,%rbx
40477d je 4047c4 <StringCreation(benchmark::State&)+0x64>
40477f test %bpl,%bpl
404782 jne 4047c4 <StringCreation(benchmark::State&)+0x64>
404784 lea 0x18(%rsp),%r15
404789 nopl 0x0(%rax)
404790 20.53% mov %r15,0x8(%rsp)

transform

```
std::vector<int> numbers(10000);
std::iota(begin(numbers), end(numbers), 0);
```

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

Ranges:

```
ranges::push_back(results, numbers | ranges::view::transform(times2));
```

STL algorithms:

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

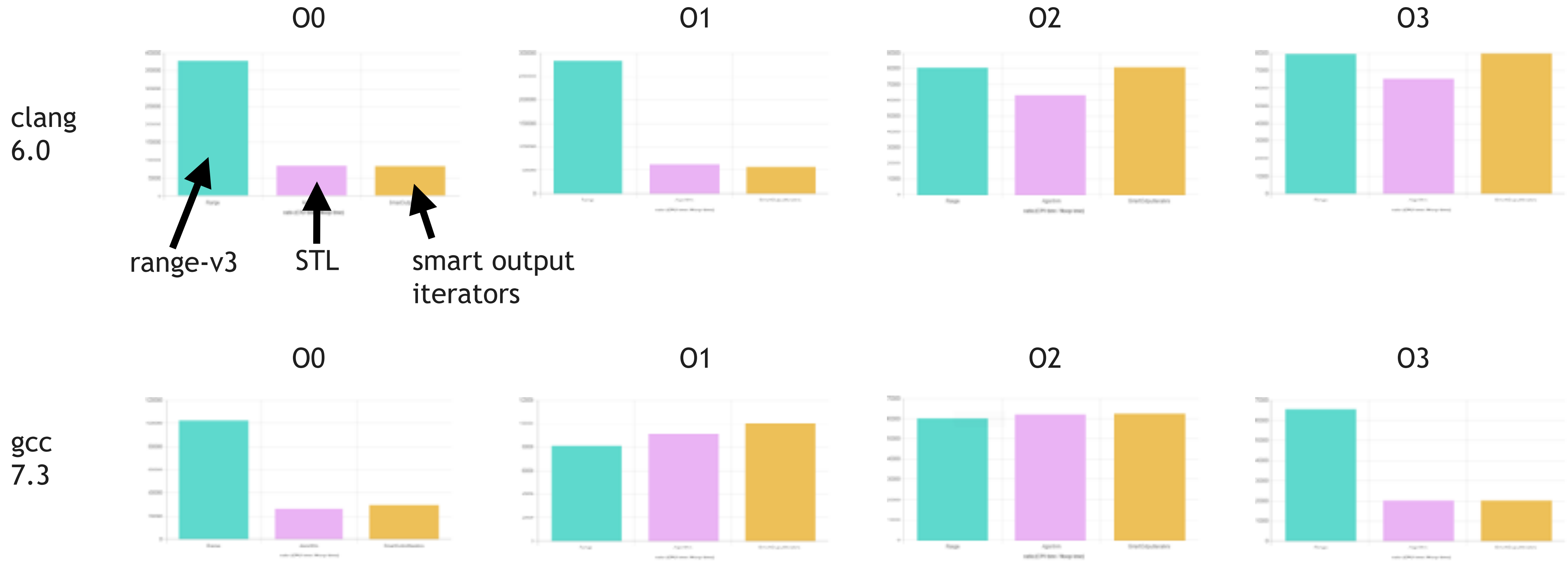
Smart output iterators:

```
auto toTimes2 = make_output_transformer(times2);
std::copy(begin(numbers), end(numbers), toTimes2(back_inserter(results)));
```

MUREX™
Financial Software

Fluent{C++}

transform



http://quick-bench.com/F-Z9zcZHYAMQw7aV3B3KAs9vNWg

filter then transform

```cpp
std::vector<int> numbers(10000);
std::iota(begin(numbers), end(numbers), 0);
```

```cpp
int times2(int x)
{
    return x * 2;
}
```

```cpp
bool isEven(int x)
{
    return x % 2 == 0;
}
```

Ranges:

```cpp
ranges::push_back(results, numbers | ranges::view::filter(isEven) | ranges::view::transform(times2))
```

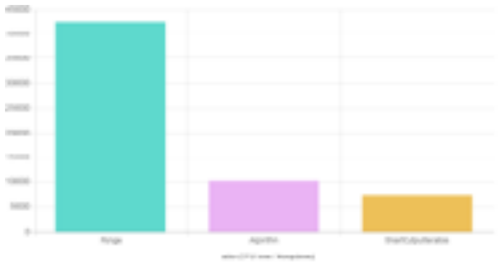
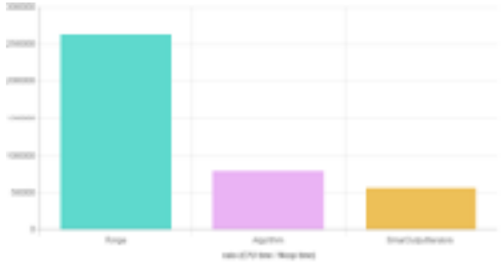
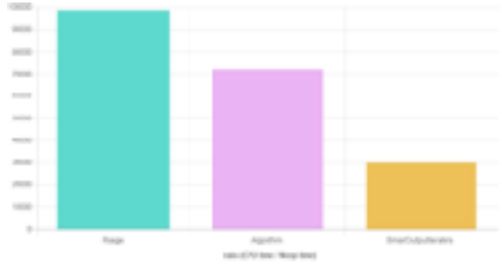
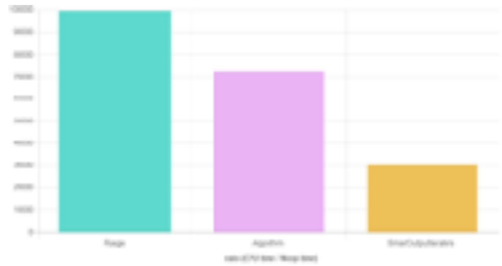
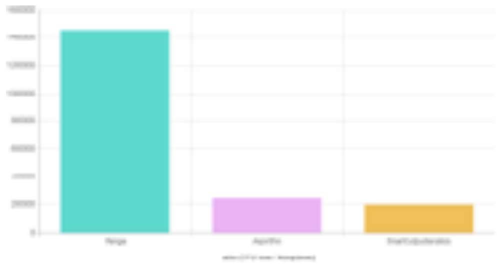
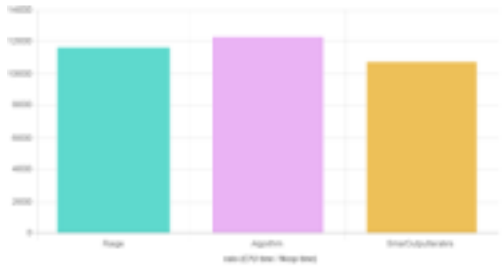
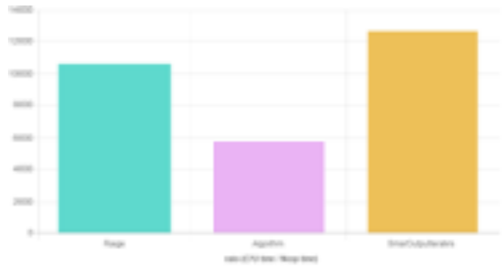
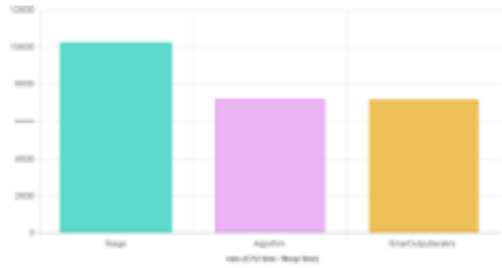
STL algorithms:

```cpp
std::vector<int> filteredNumbers;
std::copy_if(begin(numbers), end(numbers), back_inserter(filteredNumbers), isEven);
std::transform(begin(filteredNumbers), end(filteredNumbers), back_inserter(results), times2);
```

Smart output iterators:

```cpp
auto toTimes2 = make_output_transformer(times2);
auto toIsEven = make_output_filter(isEven);
std::copy(begin(numbers), end(numbers), toIsEven(toTimes2(back_inserter(results))));
```

filter then transform

	O0	O1	O2	O3
clang 6.0				
gcc 7.3				

http://quick-bench.com/p_MbUDJYd5LqJZ0PhCvGmRqwLeY

transform then filter

```
std::vector<int> numbers(10000);
std::iota(begin(numbers), end(numbers), 0);
```

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

```
bool isMultiple4(int x)
{
    return x % 4 == 0;
}
```

Ranges:

```
ranges::push_back(results, numbers | ranges::view::transform(times2) | ranges::view::filter(isMultiple4));
```

STL algorithms:

```
std::vector<int> transformedNumbers;
std::transform(begin(numbers), end(numbers), back_inserter(transformedNumbers), times2);
std::copy_if(begin(transformedNumbers), end(transformedNumbers), back_inserter(results), isMultiple4);
```

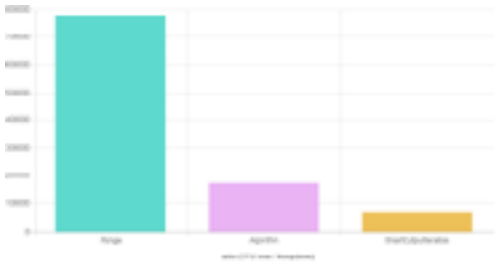
Smart output iterators:

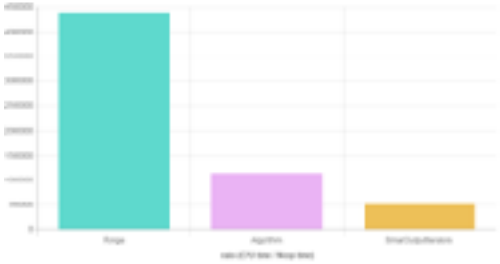
```
auto toIsMultipleOf4 = make_output_filter(isMultiple4);
auto toTimes2 = make_output_transformer(times2);
std::copy(begin(numbers), end(numbers), toTimes2(toIsMultipleOf4(back_inserter(results))));
```

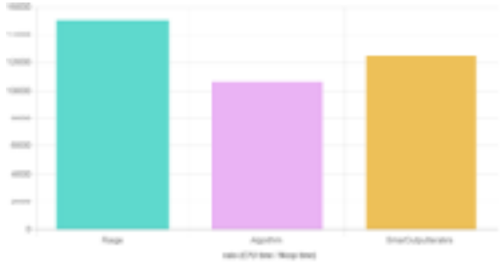
transform then filter

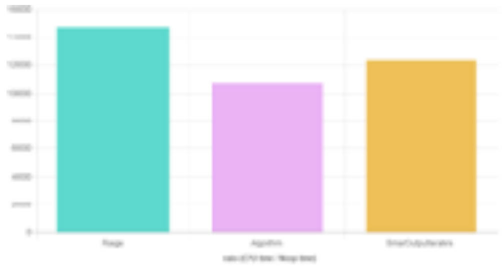
O0 O1 O2 O3

clang 6.0



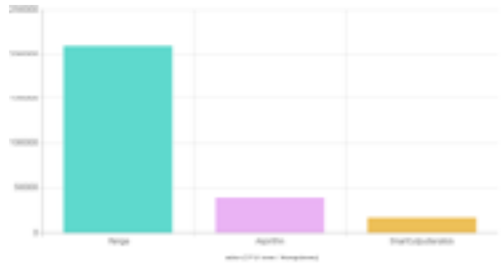


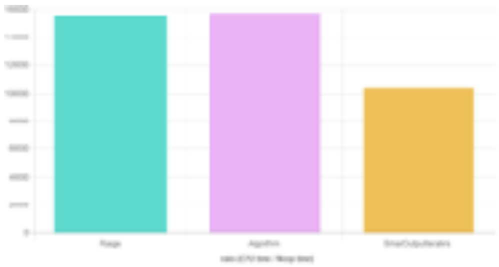


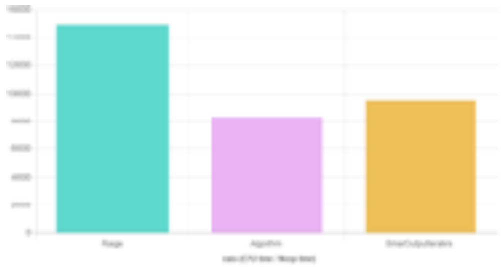


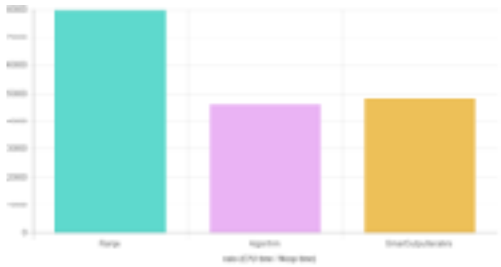
O0 O1 O2 O3

gcc 7.3









http://quick-bench.com/9FrwL28I2DJDDrMIcBUNALgdi9A

CONCLUSION

There isn't just the STL algorithms to manipulate collections.

Ranges and Smart output iterators expand your possibilities.

We need to choose the most adapted combination to make our code expressive.

github.com/JoBoccara/smart-output-iterators

@JoBoccara

Fluent {C++}

EXPRESSIVE CODE IN C++

fluentcpp.com