RANGE PREDICATES WITH THE STL

- **Predicates on ranges with the STL**
- **How to Check If a String Is a Prefix of Another One in C++**
- **Understanding the implementation of `std::is_permutation`**
- **How to Use `is_permutation` on Collections of Different Types**

Fluent {C++}

Predicates on ranges with the STL

In this mini-ebook, we examine algorithms that can be used in a variety of contexts but that have one thing in common: they return a boolean characteristic of one or several ranges.

Let's start with the *_of series.

The *_of series

The STL provides 3 algorithms that indicates whether all, some, or none of the elements of a range satisfy a given condition. The condition is itself expressed by a predicate, that is to say a function pointer (or object) that take an element of the range and returns a `bool`.

These 3 algorithms are:

- `std::all_of`: checks whether **all** of the elements in the range satisfy the given condition. It returns `true` if the range is empty, so its semantic is more precisely checking if no element does not satisfy the condition.
- `std::any_of`: checks whether **any** one of the elements in the range satisfies the given condition. It returns `false` if the range is empty.
- `std::none_of`: checks whether **no** element in the range satisfy the given condition. It returns `true` if the range is empty.

This is it for the STL, but Boost goes a bit further and proposes the following algorithm:

- `boost::algorithm::one_of`: checks whether **exactly one** element in the range satisfies the given condition. It (quite expectably) returns `false` if the range is empty.

Boost also provides "***_equal**" versions of each of the above algorithms, that accept a range and a value, and have the same behaviour as their native counterpart, with the condition being that the element is equal to the passed value. The comparison is done with `operator==` and cannot be customised.

- `boost::algorithm::`**`all_of_equal`**: takes a range and a value, and checks if all elements of the range are equal to the value.
- `boost::algorithm::`**`any_of_equal`**: takes a range and a value, and checks if any one element of the range is equal to the value.
- `boost::algorithm::`**`none_of_equal`**: takes a range and a value, and checks if any no element of the range is equal to the value.
- `boost::algorithm::`**`one_of_equal`**: takes a range and a value, and checks if any exactly one element of the range is equal to the value.

And in the case of an empty range, they behave the same way as their native counterparts.

std::equal

`std::equal` can be used to compare 2 ranges, checking if elements are respectively equal (comparison is done with `operator==` or with a custom comparator). Note that `std::equal` takes a 1.5-Range, meaning that the first range is indicated by a begin and an end iterator, while the second range misses the end iterator:

```
template<template InputIterator1, template InputIterator2 >
bool equal(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2);
```

So the algorithm goes on until the end of the 1st range and performs comparisons with the 2nd range even if it shorter, because it just doesn't know how long the second range is.

For `std::equal`, this is both unnatural and dangerous:

- this is **unnatural**, because if the 1st range has, say, N elements, `std::equal` returns `true` as long as the first N elements of the 2nd range are equal to the N elements of the 1st range, and **even if the 2nd range has more elements** than the 1st range.
- this is **dangerous**, because if the 2nd range is *shorter* than the 1st range, the algorithm will go **past its end**, leading to undefined behaviour.

Starting from C++14 this is corrected, with new overloads of `std::equal` taking 2 complete ranges with both begins and ends.

Checking for permutations

Say that we have two collections. How do you determine if one is a permutation of the other? Or, said differently, if one contains the same elements as the other, even if in a different order?

To do that, the STL offers `std::is_permutation`.

For instance, given the following collections:

```
std::vector<int> v1 = {1, 2, 3, 4, 5};
std::vector<int> v2 = {4, 2, 3, 1, 5};
std::vector<int> v3 = {2, 3, 4, 5, 6};
```

Calling `std::is_permutation` this way:

```
std::is_permutation(v1.begin(), v1.end(),
                    v2.begin(), v2.end());
```

returns `true`, while

```
std::is_permutation(v1.begin(), v1.end(),
                    v3.begin(), v3.end());
```

returns `false`, because the elements of `v3` are different from those of `v1`.

Before C++14, `std::is_permutation` had an 1.5-Range interface, that is to say that it accepted a begin and end for the first range, and **only a begin** iterator for the second one:

```
std::is_permutation(v1.begin(), v1.end(),
                    v2.begin());
```

So if the second collection was smaller than the first one, the algorithm would happily query it past its end until it gets to the end of the first one, thus causing underfined behaviour. The consequence was that you must be sure that the second collection was at least as big as the first one.

But this has been corrected in C++14, that adds the overload taking a begin *and* an end iterator for both collections.

`std::is_permutation` compares elements with **operator==**, and provides an overload that accepts custom comparators.

The algorithmic complexity of std::is_permutation

`std::is_permutation` has a complexity of “at most O(n^2)”.

That can sound surprising: indeed, the algorithms of the STL are known to be implemented with the best possible algorithmic complexity. And it seems like we could do better than quadratic complexity, can’t we?

It turns out we can, but at the expense of extra memory allocation, and if you’re interested to read more about that I suggest you take a look at Quentin’s article Lost in Permutation Complexity. So it’s a trade-off between CPU and memory. Sounds familiar, doesn’t it?

A use case for std::is_permutation

Consider a function that returns a collection of values (or produces it via an output iterator), but does not specify in which order those elements are positioned inside the collection.

How would you write a unit test for this function?

You can’t test an `EXPECT_EQ` between the expected output and the actual one, since we don’t know what the output should be equal to exactly, since we don’t know the order of its elements.

Instead, you can use `std::is_permutation`:

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

std::vector<int> results = f();

EXPECT_TRUE(std::is_permutation(begin(expected), end(expected),
                                begin(results), end(results)));
```

This way you can express that you expect the function `f` to return 1, 2, 3, 4 and 5, but in any order.

std::mismatch and std::lexicographical_compare

These 2 algorithms let you implement some sort of ordering on ranges, which you can use to compare 2 ranges.

More specifically:

`std::mismatch` compares respective elements of its 2 input ranges starting from their beginning, and returns the first place where they differ, in the form of an `std::pair` of iterators: the first element of the pair is an iterator to the first mismatching element in the 1st range, and second element of the pair is an iterator to the first mismatching element in the 2nd range.

It performs comparisons with `operator==` (or a custom comparator).

```
template<typename InputIt1, typename InputIt2, typename
BinaryPredicate>
std::pair<InputIt1,InputIt2>
    mismatch(InputIt1 first1, InputIt1 last1,
             InputIt2 first2,
             BinaryPredicate p);
```

Note that `std::mismatch` also suffers from the 1.5-Range problem, so make sure you pass the shorter range first. This can be cumbersome if you do use it to make comparisons. But just as for `std::equal`, the 1.5-Range problem is solved for `std::mismatch` starting from C++14.

`std::lexicographical_compare` actually provides an order on ranges, and operates the same way as a **dictionary** would provide an order on strings, hence its name. It compares elements two by two with `operator<` (or a custom comparator).

```
template<typename InputIt1, typename InputIt2, typename Compare>
bool lexicographical_compare(InputIt1 first1, InputIt1 last1,
                             InputIt2 first2, InputIt2 last2,
                             Compare comp );
```

`std::lexicographical_compare` takes 2 full ranges so it does not have the 1.5-Range problem.

`std::lexicographical_compare` can be quite handy for allowing a natural and easy to code order on classes wrapping a container. For instance, say when treating CSV-like data that we design an `Entry` class that represents all the pieces of data separated by commas on a given line in the CSV file:

```
class Entry
{
public:
    // ...Entry interface...
    bool operator<(const Entry& other)
    {
        return std::lexicographical_compare(begin(data_),
end(data_),
                                                 begin(other.data_),
end(other.data_));
    }
private:
    std::vector<std::string> data_;
};
```

This allows for an easy sort of entries in a natural way, which gives access to rapid searching and related functionalities (insertion, and so on). It also makes `Entry` compatible with sorted associative containers like `std::map`, `std::set` et alii.

How to Check If a String Is a Prefix of Another One in C++

The simple operation of checking if a string is a prefix of another one is not standard in C++. We will implement it step by step, and at the end of this article you'll find the complete implementation ready to paste into your code.

We will also make the code generic for checking is **any sequence is a prefix of another one**.

In C++20, the `std::string` offers this feature in its interface, with the `start_with` member function (that has been added along the `end_with` member function). Thanks to Marshall Clow for pointing this out.

Before C++20 we need to write some code ourselves. We'll also make it generic so that it applies to other sequences than `std::string`.

It's an interesting case study, because it will make us go over several aspects of writing expressive code:

- Designing a clear interface,
- Reusing standard code, with standard algorithms of the STL,
- Respecting levels of abstraction,
- Getting more familiar with the STL (with the topic of 1.5-ranges).

Let's start by designing the interface.

A "strong" interface

The role of our function is to check if a string is a prefix of another string, and this information should be displayed in the prototype. We can achieve that by naming out the function `isPrefix`, and let the parameters express that the function needs two string to operate. Indeed, to make concise names, no need to repeat the info of the parameters in the function name.

There is something we need to pay special attention to in this interface though. It takes two strings: one is the prefix, and the other one is the larger string that we're checking if it starts with that prefix. And we need to be very clear which is which.

Calling them `s1` or `s2` it would be confusing for a user of the interface, because they wouldn't know which is which. The least we can do is to show the roles of the two parameters through their names:

```
bool isPrefix(std::string const& prefix, std::string const& text);
```

It shows which parameters is expected, when writing code that uses `isPrefix`. But there is still a chance of getting it wrong and mixing up the two parameters by accident.

This sort of accident can happen if you're not paying too much attention (say, if you've just been interrupted) or if the interface changes in a branch and you're working in another branch, and the two get merged without noticing the silent collision, for example.

Also, at call site you can't tell which string is tested for being the prefix of the other one:

```
isPrefix(myFirstString, mySecondString); // which one is the prefix
of the other?
```

To help with those issues we can use **strong types**: putting the information not only in the parameter name, but also in the **parameter type**.

There are several ways to do strong typing in C++. We could use the NamedType library, but for such a simple case a `struct` will do the job:

```
struct Prefix { std::string const& value; };
struct Text { std::string const& value; };
bool isPrefix(Prefix prefix, Text text);
```

You could prefer to make the const and reference attributes show in the strong type names:

```
struct PrefixConstRef { std::string const& value; };
struct TextConstRef { std::string const& value; };

bool isPrefix(PrefixConstRef prefix, TextConstRef text);
```

There is more info in the interface but the call site becomes more verbose:

```
isPrefix(PrefixConstRef(myFirstString),
TextConstRef(mySecondString));
```

How do you feel about this trade-off? I prefer the first option, for the simpler call site, but would be interested in knowing your opinion. Don't hesitate to leave a comment.

Now we have our interface!

```
struct Prefix { std::string const& value; };
struct Text { std::string const& value; };

bool isPrefix(Prefix prefix, Text text);
```

Let's now write the implementation of the `isPrefix` function.

Reusing code for the implementation

There is no `isPrefix` in the C++ standard library, but since it's such a natural thing to do, there must something not too far from it.

And there is: the `std::mismatch` STL algorithm will do most of the work of `isPrefix`.

std::mismatch

`std::mismatch` is one of the STL algorithms that query a property on **two ranges**. It walks down the two ranges while their elements are equal, and stops whenever they start to differ. The algorithm then returns the two positions in the respective ranges (in the form of a pair of iterators), at those places where they start to differ:

Here is its prototype:

```
template<typename InputIterator1, typename InputIterator2>
std::pair<InputIterator1, InputIterator2> mismatch(InputIterator1
first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2);
```

Checking if a string is a prefix of another one is a special case of what `std::mismatch` does: it comes down to checking that the first position where they start to differ is the **end of the prefix string**.

So here is a possible implementation for `isPrefix`:

```
bool isPrefix(Prefix prefix, Text text)
{
    auto const differingPositions =
std::mismatch(begin(prefix.value), end(prefix.value),
begin(text.value), end(text.value));
    return differingPositions.first == end(prefix.value);
}
```

Raising the level of abstraction to ranges

This is a concise implementation, but we could go further and get rid of the iterators. We can wrap `std::mismatch` in an interface that expect the ranges (here, the strings) themselves.

```
namespace ranges
{
    template<typename Range1, typename Range2>
    std::pair<typename Range1::const_iterator, typename
Range2::const_iterator> mismatch(Range1 const& range1, Range2 const&
range2)
    {
        return std::mismatch(range1.begin(), range1.end(),
range2.begin(), range2.end());
    }
}
```

Using it, the code of `isPrefix` becomes simpler:

```
bool isPrefix(Prefix prefix, Text text)
{
    auto const differingPositions = ranges::mismatch(prefix.value,
text.value);
    return differingPositions.first == end(prefix.value);
}
```

The problem of 1.5 ranges

The STL overload of `std::mismatch` that we used took the two ranges in the form of a begin and an end iterator. This is the C++14 version of `std::mismatch`. And before C++14 the only available overload of `std::mismatch` was:

```
template<typename InputIterator1, typename InputIterator2>
std::pair<InputIterator1, InputIterator1> mismatch (InputIterator1
first1, InputIterator1 last1, InputIterator2 first2);
```

Note that this overload doesn't take the last of the second range! It expects that the second be at least as long at the first one, and carries on until reaching the end of the first range (or two differing values).

The dramatic consequence is that if the first range is longer than the second one, `std::mistmatch` can read past the end of the second collection. And you don't want that to happen because this is undefined behaviour (typically a crash of the application here).

But on the other hand, you don't want to deal with this algorithm problem in the code of `isPrefix` either.

The range overload is a convenient place to put that logic, as it has access to the size of the ranges and can compare them. Indeed, if the second ranges happens to be shorter than the first one (the case where old `std::mismatch` doesn't work), then we can swap the parameters:

```
namespace ranges
{
    template<typename Range1, typename Range2>
    std::pair<typename Range1::const_iterator, typename
Range2::const_iterator> mismatch(Range1 const& range1, Range2 const&
range2)
    {
        if (range1.size() <= range2.size())
        {
            return std::mismatch(range1.begin(), range1.end(),
range2.begin());
        }
        else
        {
            auto const invertedResult =
std::mismatch(range2.begin(), range2.end(), range1.begin());
```

```
            return std::make_pair(invertedResult.second,
invertedResult.first);
        }
    }
}
```

Checking for prefix in any sequence

Why just limit our code to `std::string`s? It makes sense just as well to check if a sequence of elements of any type, not just `char`s, is a prefix of another one.

So let's make our code generic to support any type of elements. Starting with the strong types:

```
template<typename T>
struct Prefix { T const& value; };

template<typename T>
struct MainSequence { T const& value; };
```

Before C++17, we need to make helper functions to deduce templates types (in C++17 the constructor is able to deduce the template types):

```
template<typename T>
Prefix<T> prefix(T const& value)
{
    return Prefix<T>{value};
}

template<typename T>
MainSequence<T> mainSequence(T const& value)
{
    return MainSequence<T>{value};


}
```

We can now make `isPrefix` generic too:

```
template<typename T, typename U>
bool isPrefix(Prefix<T> prefix, MainSequence<U> mainSequence)
{
    auto const differingPositions = ranges::mismatch(prefix.value,
mainSequence.value);
    return differingPositions.first == end(prefix.value);
}
```

And use it with other sequences than strings:

```
std::vector<int> v1{1, 2, 3, 4, 5};
std::vector<int> v2{1, 2, 3, 4, 5, 6, 7, 8, 9, 0};

auto isV1PrefixOfV2 = isPrefix(prefix(v1), mainSequence(v2));
```

Here is all the code put together:

```
template<typename T>

struct Prefix { T const& value; };

template<typename T>
struct MainSequence { T const& value; };

template<typename T>
Prefix<T> prefix(T const& value)
{
    return Prefix<T>{value};
}
template<typename T>
MainSequence<T> mainSequence(T const& value)
{
    return MainSequence<T>{value};
}

namespace ranges
{
    template<typename Range1, typename Range2>
    std::pair<typename Range1::const_iterator, typename
Range2::const_iterator> mismatch(Range1 const& range1, Range2 const&
range2)
    {
        if (range1.size() >= range2.size())
        {
            return std::mismatch(range1.begin(), range1.end(),
range2.begin());
        }
        else
        {
            auto const invertedResult =
std::mismatch(range2.begin(), range2.end(), range1.begin());
            return std::make_pair(invertedResult.second,
invertedResult.first);
        }
    }
}

template<typename T, typename U>
bool isPrefix(Prefix<T> prefix, MainSequence<U> mainSequence)
```

```
{
    auto const differingPositions = ranges::mismatch(prefix.value,
mainSequence.value);
    return differingPositions.first == end(prefix.value);
}
```

If you have any comment on this case study, your feedback will be welcome!

Understanding the implementation of std::is_permutation

Knowing your STL algorithms is a good thing. And knowing what's inside of them is a great way to go further in their study.

In that spirit, let's dig into the implementation of `std::is_permutation`. It's a nice algorithm to study, as it can be implemented by using other STL algorithms and it has some interesting subtleties. But nothing impossibly complicated.

As a reminder on algorithms on permutations, `is_permutation` takes two collections (in the form of iterators of begin and end), and returns a `bool`. This `bool` indicates whether the two collections have the same contents, but possibly not in the same order.

A naive (but wrong) implementation of `is_permutation`

The complexity of `is_permutation`, as described by the C++ standard, is O(n^2), where n is the size of the first collection.

As a side note, there are ways to implement `is_permutation` with a better algorithmic complexity, at the expense of other parameters – check out Quentin Duval's great analysis on the topic if you want to read more about that. But here, we focus on a standard-like implementation.

With a quadratic complexity, the first idea that comes to mind is to go over the first collection, and check for each element to see if it is part of the other one:

```
  template<typename ForwardIterator1, typename ForwardIterator2>
bool my_is_permutation(ForwardIterator1 first1, ForwardIterator1
last1,
                       ForwardIterator2 first2, ForwardIterator2 last2)
{
    for (auto current1 = first1; current1 != last1; ++current1)
    {
        if (std::find(first2, last2, *current1) == last2)
        {
            return false;
        }
    }
    return true;
}
```

If we test it with two collections that are permutations of each other:

```
std::vector<int> v1 = {1, 2, 3, 4, 5};
std::vector<int> v2 = {3, 2, 5, 4, 1};
std::cout << my_is_permutation(begin(v1), end(v1), begin(v2),
end(v2)) << '\n';
```

This outputs:

```
1
```

All good.

Now let's test it with two collections that are not permutations of each other:

```
std::vector<int> v1 = {1, 2, 3, 4, 5};
std::vector<int> v3 = {3, 2, 6, 4, 1};
std::cout << my_is_permutation(begin(v1), end(v1), begin(v3),
end(v3)) << '\n';
```

It now outputs:

```
0
```

Still OK. Is it a correct implementation then?

libc++ implementation

Let's compare it with the one from libc++, the implementation of the standard library used by clang:

```
template<class _BinaryPredicate, class _ForwardIterator1, class
_ForwardIterator2>
_LIBCPP_CONSTEXPR_AFTER_CXX17 bool
__is_permutation(_ForwardIterator1 __first1, _ForwardIterator1
__last1,
                 _ForwardIterator2 __first2, _ForwardIterator2
__last2,
                    _BinaryPredicate __pred,
                    forward_iterator_tag, forward_iterator_tag )
{
//  shorten sequences as much as possible by lopping of any equal
prefix
    for (; __first1 != __last1 && __first2 != __last2; ++__first1,
(void) ++__first2)
        if (!__pred(*__first1, *__first2))
            break;
    if (__first1 == __last1)
        return __first2 == __last2;
    else if (__first2 == __last2)
        return false;

    typedef typename
iterator_traits<_ForwardIterator1>::difference_type _D1;
   _D1 __l1 = _VSTD::distance(__first1, __last1);

    typedef typename
iterator_traits<_ForwardIterator2>::difference_type _D2;
    _D2 __l2 = _VSTD::distance(__first2, __last2);
    if (__l1 != __l2)
        return false;

    // For each element in [f1, l1) see if there are the same number
of
    //    equal elements in [f2, l2)
    for (_ForwardIterator1 __i = __first1; __i != __last1; ++__i)
    {
    //  Have we already counted the number of *__i in [f1, l1)?
        _ForwardIterator1 __match = __first1;
        for (; __match != __i; ++__match)
            if (__pred(*__match, *__i))
                break;
        if (__match == __i) {
            // Count number of *__i in [f2, l2)
            _D1 __c2 = 0;
            for (_ForwardIterator2 __j = __first2; __j != __last2;
++__j)
                if (__pred(*__i, *__j))
                    ++__c2;
            if (__c2 == 0)
                return false;
            // Count number of *__i in [__i, l1) (we can start with
1)
            _D1 __c1 = 1;
            for (_ForwardIterator1 __j = _VSTD::next(__i); __j !=
__last1; ++__j)
                if (__pred(*__i, *__j))
                    ++__c1;
            if (__c1 != __c2)
                return false;
        }
       }
    return true;
}
```

Wow. This looks a lot more elaborate than our naive attempt!

Our attempt can indeed be broken quite easily, with the following example:

```
std::vector<int> v1 = {1, 2, 3, 4, 5};
std::vector<int> v4 = {3, 2, 4, 4, 1};
std::cout << my_is_permutation(begin(v4), end(v4), begin(v1),
end(v1)) << '\n';
```

Which outputs:

```
1
```

It says that they are permutations of each other, while they really aren't.

So let's see what should be in the implementation of `is_permutation` to make it correct.

Implementing a correct version of `is_permutation`

The problem with our previous version of `is_permutation` is that it doesn't deal with the case of multiple occurences of the same value. What we want to check if each value in the first collection appears the same number of times in both collections, and that both collections have the same size.

We can change our algorithms in that sense:

```
template<typename ForwardIterator1, typename ForwardIterator2>
bool my_is_permutation(ForwardIterator1 first1, ForwardIterator1
last1,
                       ForwardIterator2 first2, ForwardIterator2
last2)
{
    if (std::distance(first1, last1) != std::distance(first2,
last2)) return false;

    for (auto current1 = first1; current1 != last1; ++current1)
    {
        auto const numberOfOccurencesIn1 = std::count(first1, last1, 
*current1);
        auto const numberOfOccurencesIn2 = std::count(first2, last2, 
*current1);
        if (numberOfOccurencesIn1 != numberOfOccurencesIn2)
        {
            return false;
        }
    }
    return true;
}
```

The algorithm now has a guard at the beginning, to check for the size of the two passed ranges. Then it checks that each value from the first collection is represented as many times in the second one.

This version of the algorithm passes all the previous tests (which is admittedly not enough for a test suite, we'd need at least to test for empty collections, collections of different sizes, etc. but here we focus on the algorithm rather than how to constitute the test suite – which is an equally important topic though).

Our implementation is getting more elaborate, but it's nowhere near the one of libc++! What features are missing from our implementation of `is_permutation`?

We've got the core of the algorithm right, but there are ways we can optimize it.

Discarding useless work in `is_permutation`

Our current version of `is_permutation` does way too many things. Here are a few ways to cut down some of its operations.

Similar prefix

A first thing to note is that if the two collections start with a similar sequence of elements, all there is to do is to check if their respective remainders are permutations of each other. So we can start by advancing in both collections until they start to differ.

It happens that there is an STL algorithm that does just that, and that we encountered in the predicates on ranges with the STL: `std::mismatch`. We can use it at the beginning of our algorithms:

```
template<typename ForwardIterator1, typename ForwardIterator2>
bool my_is_permutation(ForwardIterator1 first1, ForwardIterator1
last1,
                       ForwardIterator2 first2, ForwardIterator2
last2)
{
    if (std::distance(first1, last1) != std::distance(first2,
last2)) return false;

    auto const [firstDifferent1, firstDifferent2] =
std::mismatch(first1, last1, first2, last2);

    for (auto current1 = firstDifferent1; current1 != last1;
++current1)
    {
        auto const numberOfOccurencesIn1 =
std::count(firstDifferent1, last1, *current1);
        auto const numberOfOccurencesIn2 =
std::count(firstDifferent2, last2, *current1);
        if (numberOfOccurencesIn1 != numberOfOccurencesIn2)
        {
            return false;
        }
    }
    return true;
}
```

The above code uses C++17's structured bindings, but note that C++11's `std::tie` and C++98's `std::pair` can achieve an equivalent (but less elegant) result.

Counting each value only once

If our current implementation, if there are several occurences (say, `k` occurences) of the same value in the first collection, we would count for that value `k` times in both collections. We can therefore make sure that we haven't encountered this value before in the first collection:

```
template<typename ForwardIterator1, typename ForwardIterator2>
bool my_is_permutation(ForwardIterator1 first1, ForwardIterator1
last1,
                       ForwardIterator2 first2, ForwardIterator2
last2)
{
    if (std::distance(first1, last1) != std::distance(first2,
last2)) return false;

    auto const [firstDifferent1, firstDifferent2] =
std::mismatch(first1, last1, first2, last2);

    for (auto current1 = firstDifferent1; current1 != last1;
++current1)
    {
        if (std::find(firstDifferent1, current1, *current1) ==
current1)
        {
            auto const numberOfOccurencesIn1 =
std::count(firstDifferent1, last1, *current1);
            auto const numberOfOccurencesIn2 =
std::count(firstDifferent2, last2, *current1);
            if (numberOfOccurencesIn1 != numberOfOccurencesIn2)
            {
                return false;
            }
        }
    }
    return true;
}
```

Not counting a value that is not in the second collection

When we encouter a value for the first time in the first collection, we count for it in both collections. But if this value is not in the second collection, no need to count for it in the first one!

Indeed, in this case we know for sure that the two collections are not a permutation of one another. We can therefore perform that check first:

```
template<typename ForwardIterator1, typename ForwardIterator2>
bool my_is_permutation(ForwardIterator1 first1, ForwardIterator1
last1,
                       ForwardIterator2 first2, ForwardIterator2
last2)
{
    if (std::distance(first1, last1) != std::distance(first2,
last2)) return false;

    auto const [firstDifferent1, firstDifferent2] =
std::mismatch(first1, last1, first2, last2);

    for (auto current1 = firstDifferent1; current1 != last1;
++current1)
    {
        if (std::find(firstDifferent1, current1, *current1) ==
current1)
        {
            auto const numberOfOccurencesIn2 =
std::count(firstDifferent2, last2, *current1);
            if (numberOfOccurencesIn2 == 0 || numberOfOccurencesIn2
!= std::count(firstDifferent1, last1, *current1))
            {
                return false;
            }
        }
    }
    return true;
}
```

Note that this is at the expense of losing the name `numberOfOccurencesIn1` because we don't want to instantiate this value if not necessary. One way to get it back would be to explode the if statement into two consecutive if statements, but that could make the function more complex (any opinion on this?).

Not counting the beginning of the first collection

Finally, we don't need to count from the beginning of the first collection (or rather, the point where the collections start to differ). We can instead start counting from `current1`, since we checked that we haven't encountered it before.

Or even from one position after `current1` (which we know is not `last1` since that's the stopping condition of the for loop):

```
template<typename ForwardIterator1, typename ForwardIterator2>
bool my_is_permutation(ForwardIterator1 first1, ForwardIterator1
last1,
                       ForwardIterator2 first2, ForwardIterator2
last2)
{
    if (std::distance(first1, last1) != std::distance(first2,
last2)) return false;

    auto const [firstDifferent1, firstDifferent2] =
std::mismatch(first1, last1, first2, last2);

    for (auto current1 = firstDifferent1; current1 != last1;
++current1)
    {
        if (std::find(firstDifferent1, current1, *current1) ==
current1)
        {
            auto const numberOfOccurencesIn2 =
std::count(firstDifferent2, last2, *current1);
            if (numberOfOccurencesIn2 == 0 || numberOfOccurencesIn2
!= std::count(std::next(current1), last1, *current1) + 1)
            {
                return false;
            }
        }
    }
    return true;
}
```

Customizing the predicate

`is_permutation` also has an overload that accepts a custom predicate, to compare the elements of the collections together, instead of using `operator==`.

In our implementation, all the comparisons are performed by other STL algorithms. We can therefore pass the predicate along to those algorithms:

```
template<typename ForwardIterator1, typename ForwardIterator2,
typename Predicate>
bool my_is_permutation(ForwardIterator1 first1, ForwardIterator1
last1,
                       ForwardIterator2 first2, ForwardIterator2
last2,
                       Predicate pred)
{
    if (std::distance(first1, last1) != std::distance(first2,
last2)) return false;

    auto const [firstDifferent1, firstDifferent2] =
std::mismatch(first1, last1, first2, last2, pred);

    for (auto current1 = firstDifferent1; current1 != last1;
++current1)
    {
        auto equalToCurrent1 = [&pred, &current1](auto const&
value){ return pred(value, *current1); };
        if (std::find_if(firstDifferent1, current1, equalToCurrent1)
== current1)
        {
            auto const numberOfOccurencesIn2 =
std::count_if(firstDifferent2, last2, equalToCurrent1);
            if (numberOfOccurencesIn2 == 0 || numberOfOccurencesIn2
!= std::count_if(std::next(current1), last1, equalToCurrent1) + 1)
            {
                return false;
            }
        }
    }
    return true;
}
```

Going further

Our implementation is getting pretty close to the one in libc++, even though it seems shorter. The difference comes mainly from the fact that libc++ doesn't use any algorithm in its implementation and performs loops instead, which take up more space in code. I'm not sure about the reason why (perhaps to skip some function calls?).

Now that we're familiar with `is_permutation`'s implementation, we are better equiped to examine a surprising requirement the standard has on this algorithm: the two collection must have the same value types.

What consequences does this requirement have? How can we work around its constraints? This is what we'll see in the next article on `std::is_permutation`.

How to Use is_permutation on Collections of Different Types

`std::is_permutation` is an STL algorithm that checks if two collections contain the same values, but not necessarily in the same order.

We have encountered `is_permutation` in the STL algorithms on permutations, and we've seen how it was implemented. If you'd like a refresher on `std::permutation`, check out those two articles to get warmed up.

Today we're focusing on a particular requirement that the C++ standard specifies for `std::permutation`: **both collections have to contain values of the same type**.

More precisely, given the prototype of `is_permutation`:

```
template<typename ForwardIterator1, typename ForwardIterator2>
bool is_permutation(ForwardIterator1 first1, ForwardIterator1last1,
                    ForwardIterator2 first2, ForwardIterator2 last2,
                    Predicate comparator)
```

Then the value types of `ForwardIterator1` and `ForwardIterator2` must be the same.

Why is there such a requirement? Is this a problem? How can we work around it? This is what we tackle in this article.

The reason of the requirement

…is unknown to me. If you know why it's there, please let us know.

At first glance, it seems like it doesn't make sense. Indeed, if you take `std::equal` for example, you'll see that it doesn't have such a requirement. Indeed, if you pass in a custom comparison operator to `std::equal`, the algorithm is happy to use it to compare elements of potentially different types.

So why the requirement for `is_permutation`?

We can take a guess. There is something different between `std:equal` and `std::is_permutation`. If you remember the implementation of `std::is_permutation`, we had to perform comparisons between elements from the two collections, but also between elements inside of the first collection:

```
template<typename ForwardIterator1, typename ForwardIterator2,
typename Predicate>
bool my_is_permutation(ForwardIterator1 first1, ForwardIterator1
last1,
                       ForwardIterator2 first2, ForwardIterator2
last2,
                       Predicate pred)
{
    if (std::distance(first1, last1) != std::distance(first2,
last2)) return false;

    auto const [firstDifferent1, firstDifferent2] =
std::mismatch(first1, last1, first2, last2, pred);

    for (auto current1 = firstDifferent1; current1 != last1;
++current1)
    {
        auto equalToCurrent1 = [&pred, &current1](auto const&
value){ return pred(value, *current1); };
        if (std::find_if(firstDifferent1, current1, equalToCurrent1)
== current1)
        {
            auto const numberOfOccurencesIn2 =
std::count_if(firstDifferent2, last2, equalToCurrent1);
            if (numberOfOccurencesIn2 == 0 || numberOfOccurencesIn2
!= std::count_if(std::next(current1), last1, equalToCurrent1) + 1)
            {
                return false;
            }
        }
    }
    return true;
}
```

So maybe the requirement comes from the fact that the comparison function needs to also be able to compare elements from the first collection together, and making sure that both collections have the same value type makes things easier.

But whatever the reason, is this requirement a problem in practice?

What having the same value types prevents us to do

It is. Consider the following example:

```
std::vector<int> numbers = {1, 2, 42, 100, 256 };
std::vector<std::string> textNumbers = {"100", "256", "2", "1", "42"
};
```

We have two collections representing the same values, but expressed with different types. An embodying use case would be to validate user inputs (in text format) against expected inputs (in numerical format) without taking account of the order.

A more elaborate example would be a collection of values that embed a key, such as an ID, and that we'd like to compare with a collection of such IDs:

```
class Object
{
public:
    explicit Object(int ID) : ID_(ID) {}
    int getID() const { return ID_; }
private:
    int ID_;
};
```

We would like to write a piece of code like this:

```
std::vector<Object> objects = { Object(1), Object(2), Object(3),
Object(4), Object(5) };
std::vector<int> IDs = {4, 5, 2, 3, 1};

auto const sameIDs = std::is_permutation(begin(objects),
end(objects),
                                         begin(IDs), end(IDs),
                                         compareObjectWithID);
```

But there are two problems with this code:

- Problem 1:`std::is_permutation` is not allowed to take two collections of different value types,
- Problem 2: even if it was, how do we write the function `compareObjectWithID`?

Alternatively, we could make a `transform` of the `objects` into a new collection of `keys`. But let's say that we don't want to instantiate a new collection and burden our calling code with it.

Checking for a permutation on different types

To solve problem 1, one way is to use a custom implementation, like the one provided at the beginning of this post.

It's a sad solution, because it prevents us from using the standard implementation of `std::is_permutation`. And what makes it even sadder is that the standard implementations I've checked produced the correct result anyway.

But the C++ standard forbids it, so using `std::is_permutation` with elements of different types is technically undefined behaviour. We don't want to go down that road.

So let's assume that we use our own implementation of `is_permutation`. How do we implement a comparison function that works on different types? How do we solve Problem 2?

Indeed, notice that just comparing the two different types in the function is not enough. For example, if we use the following comparison function:

```
bool compareObjectWithID(int ID1, Object const& object2)
{
    return ID1 == object2.getID();
}
```

The problem is that the algorithm can call the predicates with different combinations of types: it can be with one `Object` and one `int`, or with two `Objects` for example. So to be on the safe side, we'd like to cram the 4 possible combinations of `int` and `Object` into the comparison function.

How do we cram several functions into one? With the double functor trick!

Or rather here, it would be the *quadruple* functor trick:

```
struct CompareObjectWithID
{
    bool operator()(int ID1, int ID2)
    {
        return ID1 == ID2;
    }
    bool operator()(int ID1, Object const& object2)
    {
        return ID1 == object2.getID();
    }
    bool operator()(Object const& object1, int ID2)
    {
        return (*this)(ID2, object1);
    }
    bool operator()(Object const& object1, Object const& object2)
    {
        return object1.getID() == object2.getID();
    }
};
```

We can use it this way:

```
td::vector<Object> objects = { Object(1), Object(2), Object(3),
Object(4), Object(5) };
std::vector<int> IDs = {4, 5, 2, 3, 1};

auto const sameIDs = my_is_permutation(begin(objects), end(objects),
                                       begin(IDs), end(IDs),
                                       CompareObjectWithID{}) <<
'\n';
```

Thoughts?

All this lets us perform a check for permutations on two collections with different value types. But if you have a different view on this subject, I'd be glad to hear it.

Do you know the reason for the requirement on `is_permutation` to operate on values of the same type?

Do you see a better way to work around that constraint, without resorting to creating an intermediary collection?

Did you ever encounter that need for `is_permutation`?