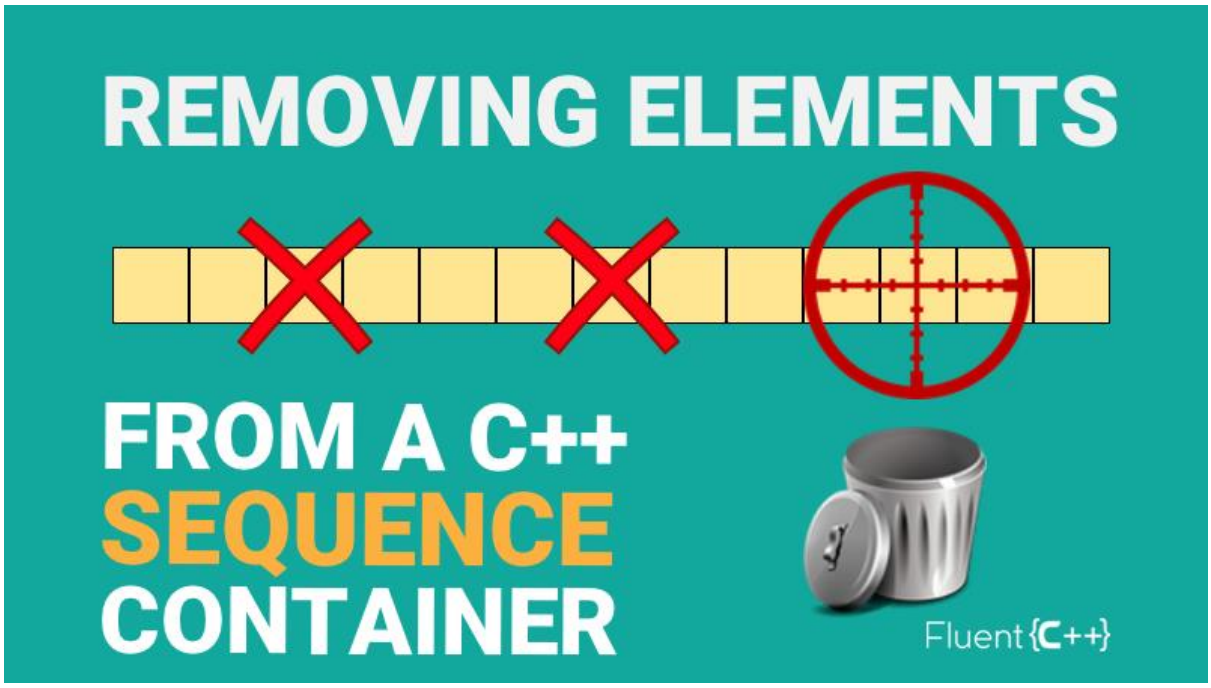
REMOVING DATA FROM C++ CONTAINERS

- **How to Remove Elements from a Sequence Container in C++**
- **How to Remove Pointers from a Vector in C++**
- **How to Remove Elements from an Associative Container in C++**
- **How to Remove Duplicates from an Associative Container in C++**

Fluent {C++}

How to Remove Elements from a Sequence Container in C++



As part of the STL Learning Resource, we're tackling today the STL algorithms that remove elements from a collection.

Removing an element from a C++ collection can't be that complicated, can it?

Well, how can I put it... It has a rich complexity, let's say.

Ok, maybe it's a little complicated.

Indeed, the approach to remove elements is very different between sequence and associative containers.

In the sequence containers, `vector` and `string` are the most commonly used. But we will cover `deque` and `list` for comprehensiveness, even if it doesn't mean that you should use them in general.

There are at least 4 ways to specify what values to remove from any container:

- Removing the elements at a given **position** (or between two given positions),
- Removing the elements equal to a certain **value**,
- Removing the elements satisfying a certain **predicate**,
- Removing the **duplicates**.

Let's see how to implement those 4 injunctions on STL sequence containers.

Removing the elements at a given position

This is the easiest way. If `c` is a sequence container, we can remove the element at the position (iterator) `position` by calling:

```
c.erase(position);
```

And to remove the element in the subrange formed by the iterators `first` and `last`, we can call:

```
c.erase(first, last);
```

Like all the ranges represented by iterators in the STL, `first` is included and `last` is not included in the subrange. `last` points to the "past-the-end" element, like the `end` iterator of a container.

Note that for `vector` and `string`, all iterators pointing to elements at and after the one removed are invalidated. Indeed, all those elements have been shifted up by the call to `erase`.

For `deque` it's a little more subtle: quoting cppreference.com, all iterators and references are invalidated, unless the erased elements are at the end or the beginning of the container, in which case only the iterators and references to the erased elements are invalidated.

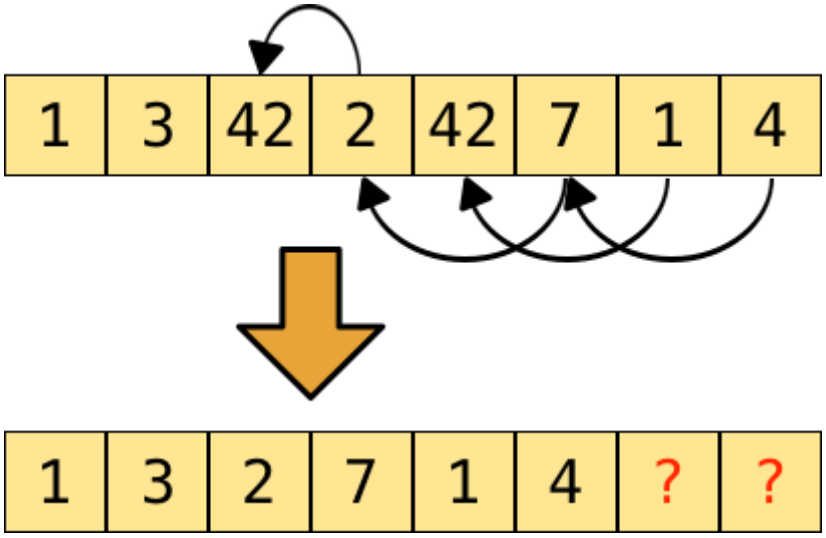
This was easy, this was warm-up. Stretch out a little and let's move on.

Removing the elements equal to a certain value

vector, deque, string

These containers don't have a method to remove a value, so we need to use the algorithm `std::remove`. This algorithm takes a range and a value to remove, and shifts up all the elements that are to be kept.

For instance, calling `std::remove` on this range of ints and with the value 42 has the following behaviour:



Note that the values of the elements left at the end of the range are unspecified. Although some implementations can leave the elements that initially were at the end of the collection, this can't be relied on.

A bit like `std::move` doesn't move and `std::forward` doesn't forward (see Effective Modern C++ item 23), `std::remove` doesn't remove. How nice is that?

Indeed, bearing in mind that, in the design of the STL, algorithms interact only with iterators, and not directly with the container, the container is not aware of the effect of the algorithm. For instance its size has not been reduced.

In order to effectively remove elements from the collection, we need to use the `erase` method that we saw in the first section of the article. For this, it is important to note that `std::remove` returns an iterator pointing to the "past-the-end" element of the range of the elements that should not be removed.

Said differently, the elements to remove are in the range defined by the iterator returned by `std::remove` and the end of the collection.

Therefore, to effectively remove values from a vector, deque or string we need to write:

```
v.erase(std::remove(begin(v), end(v), 42), end(v));
```

Wrapping the idiom

That is a C++ idiom, that you have to know if you come across it in code.

But frankly, don't you find this is a lot of code to express such a simple thing? Wouldn't you prefer to write something like:

```
v.remove(42);
```

or

```
v.erase(42);
```

But we can't add a method to `vector`. However, we can write a free function with a simple interface that takes a vector and removes some of its elements!

```
template<typename T>
void erase(std::vector<T>& vector, T const& value)
{
    vector.erase(std::remove(begin(vector), end(vector),
value), end(vector));
}
```

And while we're at it, we can add to it some overloads that operate on a `deque` and on a `string`:

```
template<typename T>
void erase(std::deque<T>& deque, T const& value)
```

```
{
    deque.erase(std::remove(begin(deque), end(deque), value),
end(deque));
}

void erase(std::string& string, char letter)
{
    string.erase(std::remove(begin(string), end(string), letter),
end(string));
}
```

I do recommend to implement those helper functions, in particular for `vector` that is the most commonly used. This will make you avoid the entanglement of iterators that comes with the standard idiom.

There even has been a proposal for the C++ standard, by Stephan T. Lavavej, to add this sort of generic function. It hasn't made it in C++17, but I suppose it still has chance to make it in a later standard.

list

Just for the sake of comprehensiveness, let's mention that to remove an element from a `list`, there is a method called `remove`:

```
l.remove(42);
```

Indeed, since it does not offer random-access iterators, using the algorithm `std::remove` on a `list` would make `list` even slower than it already is.

Removing the elements that satisfy a predicate

We've seen how to remove from a sequence container all the elements that were equal to a certain value, such as 42.

How can we remove the elements that satisfy a predicate `p`?

It's exactly the same thing, except that you need to use `remove_if` instead of `remove`.

So you just need to replace:

- `remove` by `remove_if`
- and 42 by `p`

in the previous section. Including the suggestion to write a free function `erase_if` to avoid the hord of iterators, and that `list` has a `remove_if` method.

So let's apply the Don't Repeat Yourself principle to this article and not write more about `remove_if`. Instead, let's move on to the last section: removing duplicates.

Removing duplicates from a sequence container

The STL algorithm to remove duplicate is `std::unique`.

But beware! `std::unique` only removes **adjacent duplicates**, and not duplicates in the collection as a whole. It has a linear complexity.

Other than this, `unique` is very similar to `remove`. It only squashes up the elements of the collection without having access to the container itself. So we need to call `erase` on the container to effectively remove the duplicates:

```
vector.erase(std::unique(begin(v), end(v)), end(v));
```

And, like for `remove`, a convenience function is... convenient:

```
template<typename T>
void unique(std::vector<T>& vector)
{
    vector.erase(std::unique(begin(vector), end(vector)),
end(vector));
}

template<typename T>
void unique(std::deque<T>& deque)
{
    deque.erase(std::unique(begin(deque), end(deque)), end(deque));
}
```

```
void unique(std::string& string)
{
    string.erase(std::unique(begin(string), end(string)),
end(string));
}
```

And similarly to `remove`, `std::list` has a `unique` method.

That's it for removing elements from a sequence container in C++.

Next up in our series about removing elements from a collection: removing pointers from a vector!

How to Remove Pointers from a Vector in C++

Today we have a post co-written with ***Gaurav Sehgal***, a software engineer who works with C and C++. Gaurav can be found on his *Stack Overflow profile* as well as on *LinkedIn*.

Interested in writing on Fluent C++ too? Check out our *guest posting area*!

As we saw in the article about removing elements from a sequence container, to remove elements in a vector based on a predicate, C++ uses the erase-remove idiom:

```
vector<int> vec{2, 3, 5, 2};

vec.erase(std::remove_if(vec.begin(), vec.end(), [](int i){ return i
% 2 == 0;}), vec.end());
```

Which we can wrap in a more expressive function call:

```
vector<int> vec{2, 3, 5, 2};

erase_if(vec, [](int i){ return i % 2 == 0; });
```

The resulting `vec` in both those examples contains {3, 5} after the call to the algorithm. If you'd like a refresher on the erase-remove idiom, which we use in this post, check out the dedicated article about it.

This works fine with vector of values, such as vectors of integers for example. But for **vector of pointers** this is not as straightforward, since memory management comes into play.

Removing from a vector of unique_ptrs

C++11 introduced `std::unique_ptr` along with other smart pointers, that wrap a normal pointer and takes care of memory management, by calling `delete` on the pointer in their destructors.

This allows to manipulate pointers more easily, and allows in particular to call `std::remove` and `std::remove_if` on a vector of `std::unique_ptr`s for example without a problem:

```
auto vec = std::vector<std::unique_ptr<int>>{};
vec.push_back(std::make_unique<int>(2));
vec.push_back(std::make_unique<int>(3));
vec.push_back(std::make_unique<int>(5));
vec.push_back(std::make_unique<int>(2));
```

(for reasons outside of the scope of this post, vectors of `unique_ptr` can't use a `std::initializer_list`)

```
vec.erase(std::remove_if(vec.begin(), vec.end(), [](auto const& pi){
return *pi % 2 == 0; }), vec.end());
```

Or by wrapping the erase-remove idiom:

```
erase_if(vec, [](auto const& pi){ return *pi % 2 == 0; });
```

This code effectively removes the first and last elements of the vector, that pointed to even integers.

Note that since `std::unique_ptr` can't be copied but only moved, the fact that this code compiles shows that `std::remove_if` doesn't copy the elements of the collection, but rather moves them around. And we know that moving a `std::unique_ptr u1` into a `std::unique_ptr u2` takes the ownership of the underlying raw pointer from `u1` to `u2`, leaving `u1` with a null pointer.

As a result, the elements placed by the algorithm at the beginning of the collection (in our case the `unique_ptr` to 3 and the `unique_ptr` to 5) are guaranteed to be the sole owners of their underlying pointers.

All this handling of memory happens thanks to `unique_ptr`s. But what would happen with a vector of owning raw pointers?

Removing from a vector of owning raw pointers

First, let's note that a vector of owning raw pointers is not recommended in modern C++ (even using owning raw pointers without a vector are not recommended in modern C++). `std::unique_ptr` and other smart pointers offer safer and more expressive alternative since C++11.

But even though modern C++ is pioneering ever more, not all codebases in the world are catching up at the same pace. This makes it possible for you to encounter vectors of owning raw pointers. It could be in a codebase in C++03, or in a codebase that use modern compilers but still contains older patterns in its legacy code.

Another case where you would be concerned is if you write library code. If your code accepts a `std::vector<T>` with no assumption on type `T`, you could be called from legacy code with a vector of owning raw pointers.

The rest of this post assumes that you have to deal with vector of owning raw pointers from time to time, and that you have to remove elements from them. Then using `std::remove` and `std::remove_if` is a very bad idea.

The problem of std::remove on raw pointers

To illustrate the problem, let's create a vector of owning raw pointers:

```
auto vec = std::vector<int*>{ new int(2), new int(3), new int(5),
new int(2) };
```

If we call the usual erase-remove pattern on it:

```
vec.erase(std::remove_if(vec.begin(), vec.end(), [](int* pi){ return
*pi % 2 == 0; }), vec.end());
```

Then we end up with a memory leak: the vector no longer contains the pointers to 2, but no one has called `delete` on them.

So we may be tempted to separate `std::remove_if` from the call to `erase` in order to `delete` the pointers at the end of the vector between the calls:

```
auto firstToErase = std::remove_if(vec.begin(), vec.end(), [](int*
pi){ return *pi % 2 == 0; });
for (auto pointer = firstToErase; pointer != vec.end(); ++pointer)
   delete *pointer;
vec.erase(firstToErase, vec.end());
```

But this doesn't work either, because this creates dangling pointers. To understand why, we have to consider one of the requirements (or rather, absence of) of `std::remove` and `std::remove_if`: the elements they leave at the end of the vector are **unspecified**. It could be the elements that were there before calling the algorithm, or the elements that satisfied the predicate, or anything else.

In a particular STL implementation, the elements left at the end of the container after the call to `std::remove_if` turned out to be the ones that were there before calling the algorithm. As the vector had pointers to 2 3 5 2 before calling `std::remove`, it had pointers to 3 5 5 2 after.

For example, printing the values inside of the vector before calling `std::remove` could output this:

```
0x55c8d7980c20
0x55c8d7980c40
0x55c8d7980c60
0x55c8d7980c80
```

And after the call to `std::remove` it outputs that:

```
0x55c8d7980c40
0x55c8d7980c60
0x55c8d7980c60
0x55c8d7980c80
```

Then the innocent call to `erase` will `delete` the pointer in the 3rd position, making the one in the second position (equal to it) a dangerous dangling pointer!

What to do instead

You can use `std::stable_partition` instead of `std::remove_if`, with an inverted predicate. Indeed, `std::stable_partition` performs a partitioning of the collection based on a predicate. This means to put the elements that satisfy the predicate at the beginning, and **the elements that don't satisfy the predicate at the end**. No more equal pointers.

The paritioning here consists in putting the elements **not** to remove at the beginning, hence the need to invert the predicate:

```
std::stable_partition(vec.begin(), vec.end(), [](int* pi){ return
*pi % 2 != 0; });
```

`std::stable_partition` returns the partition point of the collection, which is the iterator to the first element that doesn't satisfy the predicate after partitioning. We therefore have to `delete` the pointers from this point and until the end of the vector. After that, we can erase the elements from the vector:

```
auto firstToRemove = std::stable_partition(vec.begin(), vec.end(),
[](int* pi){ return *pi % 2 != 0; });
std::for_each(firstToRemove, vec.end(), [](int* pi){ delete pi; });
vec.erase(firstToRemove, vec.end());
```

Another solution is to delete the pointers to remove and set them to `nullptr` and only then perform a `std::remove` on `nullptr`:

```
for(auto& pointer : vec)
{
   if (*pointer % 2 == 0)
   {
       delete pointer;
       pointer = nullptr;
   }
}
vec.erase(std::remove(vec.begin(), vec.end(), nullptr), vec.end());
```

Since the `delete`s are performed before the call to `std::remove`, there is no longer the problem with dangling pointers. But this solution only works if the vector cannot contain null pointers. Otherwise, they would be removed along with the ones set by the for loop.

Be careful around owning raw pointers

In conclusion, prefer `unique_ptr`s or other smart pointers over owning raw pointers. It will make your code simpler and more expressive.

And if you do have to work with vector of owning raw pointers, choose the right STL algorithm to correctly handle memory management!

How to Remove Elements from an Associative Container in C++

Associative containers associate keys to values, and they include:

- `std::map`, that has unique keys,
- `std::multimap`, that can have several equivalent keys,
- `std::unordered_map`, the hash map with unique keys,
- `std::unordered_multimap`, the hash map that can have several equivalent keys.

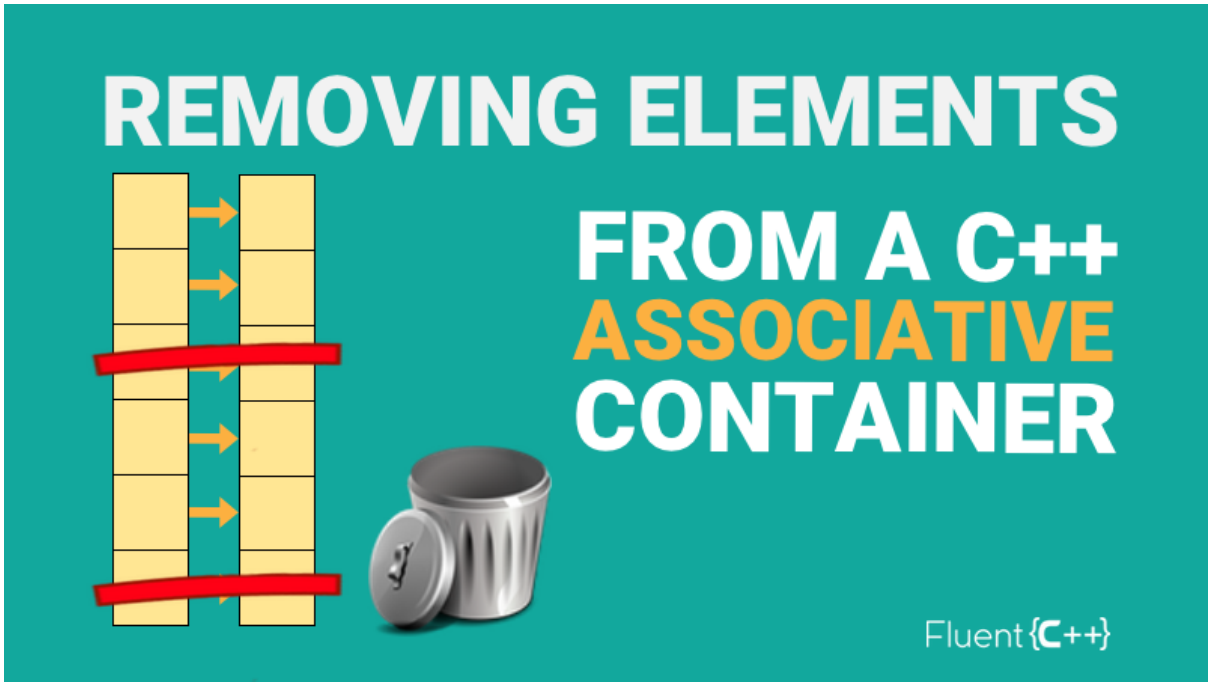
By extension, the associative containers also include sets:

- `std::set`, that has unique elements,
- `std::multiset` that can have several equivalent elements,
- `std::unordered_set`, the hash set with unique elements,
- `std::unordered_multiset`, the hash set that can have several equivalent elements.

Sets are included in associative containers because they can be seen as melting keys and values into one element.

We will answer the same 4 questions as in part one on sequence containers:

- How to remove the elements at a given **position** (or between two given positions),
- How to remove the elements equivalent to a certain **value**,
- How to remove the elements satisfying a certain **predicate**,
- How to remove the **duplicates** (this one is hairy enough to deserve its own article).



Removing the elements at a given position

As for sequence containers, removing elements from an associative container is a walk in the park if you know its position with an iterator `position`. If `a` is of any of the 8 associative containers:

```
a.erase(position);
```

removes the entry at that position.

And:

```
a.erase(first, last);
```

removes all the entries between `first` (included) and `last` (not included).

Of course, the iterators pointing to the removed elements gets invalidated, but **all other iterators to the container remain valid**. This is a difference with sequence containers.

Removing the elements equivalent to a certain key

Note that for associative containers we don't talk about "equal to a certain key" but rather "**equivalent** to a certain key". If you're not familiar with it, this subtle difference is explained in details in Custom comparison, equality and equivalence with the STL.

If you have the key of the entry you want to remove from an associative container, it's easy as pie:

```
a.erase(myKey);
```

Note that this removes all the entries whose key is equivalent to `myKey` (for the `multi` containers).

However, if you want to remove the elements of a map (or of its multi- of hash- counterparts) identified by their **value** and not their key, it's not as straightforward.

For this you need to remove the elements satisfying the **predicate** of having their value equal to something. Which leads us to the next section:

Removing the elements that satisfy a predicate

A structural difference with sequence containers

To remove elements from an sequence container according to a predicate, we used `std::remove_if`. We can't do the same thing here.

Indeed, pulling up the elements to be kept was OK in a sequence container, where the values are simply lined up one after the other (by definition of a sequence container).

But associative container have stronger constraints: they have to find keys pretty fast (in O(log(n)) for non-hash and O(1) for hash). And to achieve this, they structure the data in more complex ways, typically in a tree for non-hash containers, and in a table where exact positions matter, for hash containers.

So we can't just shuffle the elements like `std::remove_if` does, otherwise we would break the internal structure. So **we have to play along with the interface**. And what we get in the interface is the `erase` method that we've seen above.

Playing along with the interface

The general idea to remove elements according to a predicate is to iterate over the container, check the predicate on each element and remove those that return `true`. But the problem is, how to iterate and remove elements at the same time?

Indeed, consider the naive version of such an iteration:

```
template<typename AssociativeContainer, typename Predicate>
void erase_if(AssociativeContainer& container, Predicate
shouldRemove)
{
    for (auto it = begin(container); it != end(container); ++it)
    {
        if (shouldRemove(*it))
        {
            container.erase(it);
        }
    }
}
```

Note that this is one of the very rare cases where we don't know more about the iterators than that they are iterators. In other cases, I consider `it` to be one of the 7 names we should never see in code.

Anyway, consider line 8:

```
container.erase(it);
```

This has the effect of invalidating `it`. Then look at the end of line 4:

```
for (auto it = begin(container); it != end(container); ++it)
```

We do `++it` right after `it` has been invalidated. This causes undefined behaviour.

Juggling with iterators

We need to find a way to increment the iterator *before* erasing it. For this we have several options. In C++98 we can use the post-fix increment operator that will first increment the iterator and then pass a copy of the non-incremented iterator to `erase`:

```
template<typename AssociativeContainer, typename Predicate>
void erase_if(AssociativeContainer& container, Predicate
shouldRemove)
{
    for (auto it = begin(container); it != end(container); /*
nothing here, the increment in dealt with inside the loop */ )
    {
        if (shouldRemove(*it))
        {
            container.erase(it++);
        }
        else
        {
            ++it;
        }
    }
}
```

But juggling with iterators is not much less dangerous than juggling with knives. Or with torches. In C++11 we get a less risky implementation because `erase` returns the iterator following the removed elements. We can then rewrite the code this way:

```
template<typename AssociativeContainer, typename Predicate>
void erase_if(AssociativeContainer& container, Predicate
shouldRemove)
{
    for (auto it = begin(container); it != end(container); /*
nothing here, the increment in dealt with inside the loop */ )
    {
        if (shouldRemove(*it))
        {
            it = container.erase(it);
        }
        else
        {
            ++it;
        }
    }
```

```
}
```

To make sure that this function is only used with associative containers, I suppose we will be able to use a concept when they're out (in C++20, as it seems) but in the meantime we can just write the various cases explicitly:

Click To Expand Code

C++

```
namespace details
{
    template<typename AssociativeContainer, typename Predicate>
    void erase_if_impl(AssociativeContainer& container, Predicate
shouldRemove)
    {
        for (auto it = begin(container); it != end(container); /*
nothing here, the increment in dealt with inside the loop */ )
        {
            if (shouldRemove(*it))
            {
                it = container.erase(it);
            }
            else
            {
                ++it;
            }
        }
    }
}

template<typename Key, typename Value, typename Comparator, typename
Predicate>
void erase_if(std::map<Key, Value, Comparator>& container, Predicate
shouldRemove)
{
    return details::erase_if_impl(container, shouldRemove);
}

template<typename Key, typename Value, typename Comparator, typename
Predicate>
void erase_if(std::multimap<Key, Value, Comparator>& container,
Predicate shouldRemove)
{
    return details::erase_if_impl(container, shouldRemove);
}

template<typename Key, typename Value, typename Comparator, typename
Predicate>
```

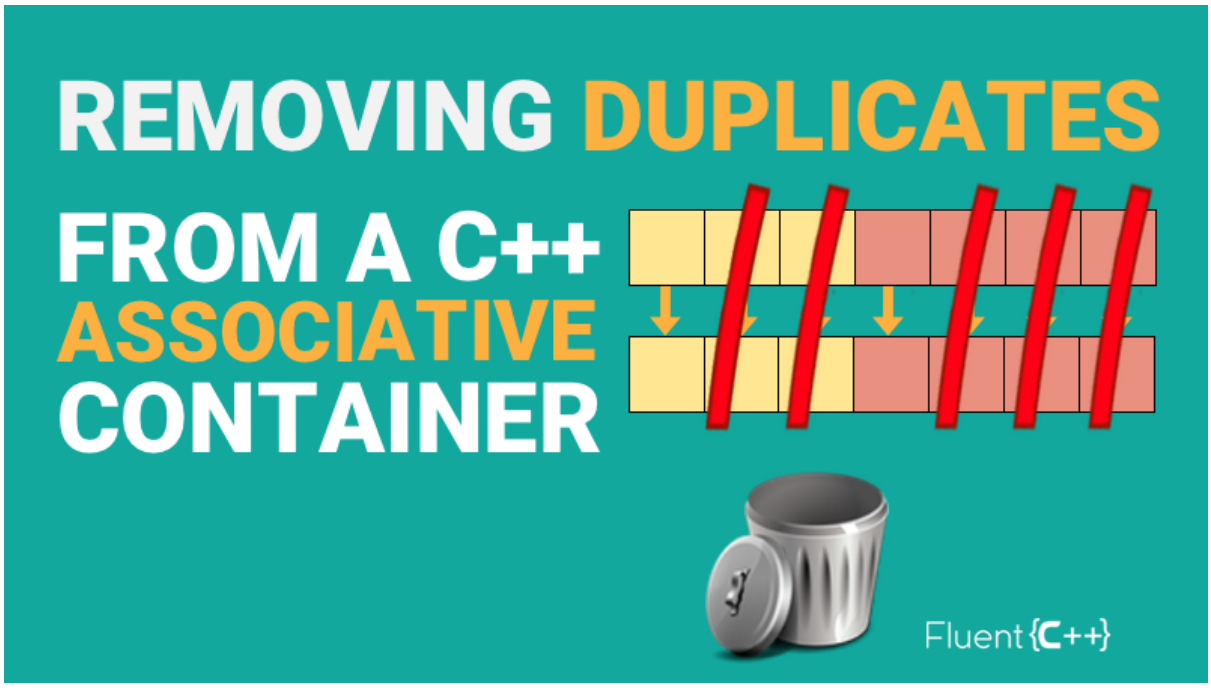
```
void erase_if(std::unordered_map<Key, Value, Comparator>& container,
Predicate shouldRemove)
{
    return details::erase_if_impl(container, shouldRemove);
}

template<typename Key, typename Value, typename Comparator, typename
Predicate>
void erase_if(std::unordered_multimap<Key, Value, Comparator>&
container, Predicate shouldRemove)
{
    return details::erase_if_impl(container, shouldRemove);
}

template<typename Key, typename Comparator, typename Predicate>
void erase_if(std::set<Key, Comparator>& container, Predicate
shouldRemove)
{
    return details::erase_if_impl(container, shouldRemove);
}

template<typename Key, typename Comparator, typename Predicate>
void erase_if(std::multiset<Key, Comparator>& container, Predicate
shouldRemove)
{
    return details::erase_if_impl(container, shouldRemove);
}

template<typename Key, typename Comparator, typename Predicate>
void erase_if(std::unordered_set<Key, Comparator>& container,
Predicate shouldRemove)
{
    return details::erase_if_impl(container, shouldRemove);
}

template<typename Key, typename Comparator, typename Predicate>
void erase_if(std::unordered_multiset<Key, Comparator>& container,
Predicate shouldRemove)
{
    return details::erase_if_impl(container, shouldRemove);
}
```

This type of generic function has been proposed by Stephan T. Lavavej for the C++ standard. The proposal hasn't made it in C++17 though. Perhaps it will be accepted along with the Ranges proposal.

Removing duplicates from an associative container

Next up in our series about removing stuff from containers in C++ we'll see how to remove duplicates from associative containers. It's a hairy topic, but one that gives a chance to get a more in-depth understanding of STL containers.

How to Remove Duplicates from an Associative Container in C++



Let's now tackle the tricky topic of removing duplicates from associative containers!

But what is a duplicate, exactly?

Removing duplicates only makes sense for the 4 associative containers that have "multi" in their name. The other don't have duplicates, by definition.

For multimap and unordered_multimap, the concept of duplicate can have several meanings: that could be two elements having the same key, but it could also be two elements having both the same key and the same value.

However, since the elements having the same key are in no specified order in the container, we can't remove (key, value) duplicates in O(n), because they may not be located next to each other. So we won't look at this latter case here. We will only look at keys to determine whether two elements are duplicates.

For sets, there is no ambiguity since keys and values are one anyway.

Note that before C++11, we didn't know which of the duplicates remain in the end. It would be the first one encountered during iteration but since they are in no specified order, this doesn't say much. In C++11, insertion adds elements at the upper bound of the range containing equivalent keys.

Also, duplicate keys don't mean the same thing between multimap and unordered_multimap: the former uses equivalence (with a "less than" semantics) and the latter uses equality (with an "equal to" semantics). And this difference is also true for multiset and unordered_multiset.

So two elements being "duplicates" can have several meanings. Let's encapsulate this under a policy: DuplicatePolicy that takes two elements and returns a bool indicating whether they are duplicates.

In all cases, the idea is the same as the one we saw when removing elements according to a predicate: iterate over the collection and remove duplicates, by being careful not to invalidate iterators.

Let's first implement the generic code using DuplicatePolicy, and then see how to implement this policy.

The traversal algorithm

Here is a possible implementation. The code is explained just afterwards:

```
template<typename AssociativeContainer, typename DuplicatePolicy>
void unique(AssociativeContainer& container, DuplicatePolicy
areDuplicates)
{
    if (container.size() > 1)
    {
        auto it = begin(container);
        auto previousIt = it;
        ++it;
        while (it != end(container))
        {
            if (areDuplicates(*previousIt, *it))
            {
                it = container.erase(it);
            }
```

```
            else
            {
                previousIt = it;
                ++it;
            }
        }
    }
}
```

Here is how this code works:

```
if (container.size() > 1)
```

The algorithm is going to consider two consecutive iterators at the same time, to compare them. We can only do this if the container has at least one element. In fact if it doesn't have at least two elements, there is no duplicate to remove anyway.

```
auto it = begin(container);

auto previousIt = it;
++it;
```

Here we make it point the second element of the container, and previousIt it to the first element.

```
while (it != end(container))
```

it is the leading iterator of the two, so we will continue until it reaches the end of the container.

```
if (areDuplicates(*previousIt, *it))
{
    it = container.erase(it);
}
else
{
    previousIt = it;
    ++it;
}
```

This structure is to avoid iterator invalidation, like when we removed according to a predicate. Note that when the element is not equivalent to the previous one, we move on the previous one to continue the traversal of the container.

How to implement the policy

We could stop here and let a client code call unique by passing a lambda that describes how to identify two duplicates. But this would present several issues:

it would burden every call site of unique with low-level and redundant information,

there would be a risk of getting the lambda wrong, especially if the container has a custom comparator.

To solve this we can provide default values for this policy, that would be correspond to the various cases.

std::multimap and std::multiset

Let's start with the non-hash multi-containers, so std::multimap and std::multiset. They both provide a method called value_comp, that returns a function comparing the keys of two elements.

Indeed, contrary to what its name suggests, value_comp for maps does not compare values. It only compares keys. Actually, it makes a lot of sense since the container has no idea how to compare the values associated to the keys. The method is called value_comp because it accepts values, and compare their keys.

To eliminate the entries with duplicate keys in a std::multimap, the policy is:

```
[&container](std::pair<const Key, Value> const& element1,
             std::pair<const Key, Value> const& element2)
             {
                 return !container.value_comp()(element1, element2)
&&
                        !container.value_comp()(element2, element1);
             }
```

Indeed, multimap and multiset use equivalence, and not equality. This means that value_comp returns a function that compares elements in the sense of being "lower than", and not "equal to". To check if two elements are duplicates, we see check that neither one is lower than the other.

So a unique function for std::multimap would be:

```
template<typename Key, typename Value, typename Comparator>
void unique(std::multimap<Key, Value, Comparator>& container)
{
    return unique(container, [&container](std::pair<const Key,
Value> const& element1,
                                          std::pair<const Key,
Value> const& element2)
                                          {
                                              return
!container.value_comp()(element1, element2) &&

!container.value_comp()(element2, element1);
                                          });
}
```

The one for multisets follows the same logic:

```
template<typename Key, typename Comparator>
void unique(std::multiset<Key, Comparator>& container)
{
    return unique(container, [&container](Key const& element1,
                                          Key const& element2)
                                          {
                                              return
!container.value_comp()(element1, element2) &&

!container.value_comp()(element2, element1);
                                          });
}
```

std::unordered_multimap and std::unordered_multiset

Let's now turn to hash multi-containers: std::unordered_multimap and std::unordered_multiset.

Before getting further, let's remember that to effectively remove duplicates from a container in one traversal those duplicates need to be next to each other. Indeed, our algorithm is in O(n). It doens't perform a full search for every value across the container (which would be O(n2)).

But unordered_multimap and unordered_multisets are... unordered! So it's not going to work, is it?

In fact it is, thanks to one property of those containers: the elements with the same keys are guaranteed to be consecutive in the iteration order. Phew.

Additionally, those containers follow a logic of equality for their keys. This means that their comparison function has the semantics of “equal to” and not “lower than”.

They offer a method to access their comparator: key_eq, that returns a function that compares keys. This method is the counterpart of key_comp in the non-hash containers.

However there is no equivalent of value_comp. There is no value_eq that would accept two elements and compare their keys. So we’ll have to make do with key_eq, and pass the keys to it ourselves. Here is the resulting code for std::unordered_multimap:

```
template<typename Key, typename Value, typename Comparator>
void unique(std::unordered_multimap<Key, Value, Comparator>&
container)
{
    return unique(container, [&container](std::pair<const Key,
Value> const& element1,
                                          std::pair<const Key,
Value> const& element2)
                                          {
                                              return
container.key_eq()(element1.first, element2.first);
                                          });
}
```

And the code for std::unordered_multiset follows the same logic:

```
template<typename Key, typename Comparator>
void unique(std::unordered_multiset<Key, Comparator>& container)
{
    return unique(container, [&container](Key const& element1,
                                          Key const& element2)
                                          {
                                              return
container.key_eq()(element1, element2);
                                          });
}
```

Here is all this code put together, with the initial generic unique function in a technical namespace:

Click To Expand Code

C++

```
#include <set>
#include <map>
#include <unordered_map>
#include <unordered_set>

namespace details
{
    template<typename AssociativeContainer, typename
DuplicatePolicy>
    void unique_associative(AssociativeContainer& container,
DuplicatePolicy areDuplicates)
    {
        if (container.size() > 1)
        {
            auto it = begin(container);
            auto previousIt = it;
            ++it;
            while (it != end(container))
            {
                if (areDuplicates(*previousIt, *it))
                {
                    it = container.erase(it);
                }
                else
                {
                    previousIt = it;
                    ++it;
                }
            }
        }
    }
}
template<typename Key, typename Value, typename Comparator>
void unique(std::multimap<Key, Value, Comparator>& container)
{
    return details::unique_associative(container,
[&container](std::pair<const Key, Value> const& element1,

std::pair<const Key, Value> const& element2)
                                                                {

return !container.value_comp()(element1, element2) &&

!container.value_comp()(element2, element1);
                                                                });
}

template<typename Key, typename Comparator>
void unique(std::multiset<Key, Comparator>& container)
{
```

```
    return details::unique_associative(container, [&container](Key
const& element1,
                                                               Key
const& element2)
                                                               {

return !container.value_comp()(element1, element2) &&

!container.value_comp()(element2, element1);
                                                               });
}

template<typename Key, typename Value, typename Comparator>
void unique(std::unordered_multimap<Key, Value, Comparator>&
container)
{
    return details::unique_associative(container,
[&container](std::pair<const Key, Value> const& element1,

std::pair<const Key, Value> const& element2)
                                                               {

return container.key_eq()(element1.first, element2.first);
                                                               });
}

template<typename Key, typename Comparator>
void unique(std::unordered_multiset<Key, Comparator>& container)
{
    return details::unique_associative(container, [&container](Key
const& element1,
                                                               Key

const& element2)
                                                               {

return container.key_eq()(element1, element2);
                                                               });
}
```

This closes off our series about removing stuff from containers in C++.

Removing elements, a simple topic? Oh no.

Removing elements, a good topic to better understand STL containers? Yes indeed.