C++ MAPS TECHNIQUES

- **Which One Is Better: Map of Vectors, or Multimap?**
- **Overview of std::map's Insertion / Emplacement Methods in C++17**
- **How to Check If an Inserted Object Was Already in a Map (with Expressive Code)**

Fluent {C++}

Which One Is Better: Map of Vectors, or Multimap?

While advising on how to make code more expressive on the SFME project, I came across an interesting case of choosing the right data structure, which I'll share with you with the permission of the authors of the projects.

We had to associate a key with several values, and perform various operations. Should we use a map of vectors, or is a multimap more appropriate? Let's see the case in more details, and compare the two solutions.

The case: an event mediator

The interface for this event system has three functions:

1- `void subscribe(EventReceiver const& receiver, EventID eventID)`

This is the method to register a receiver to a certain type of event. When this type of event occurs, the event manager notifies the `EventReceiver` on its (virtual) method `reactTo`.

2- `void emit(Event const& event) const`

This method gets called by the sender of an event when an event occurs. The method calls the `reactTo` method of all the clients that registered for its event ID.

3- `bool isRegistered(EventReceiver const& receiver) const`

At any time, we can query the event manager to know whether a given `EventReceiver` has subscribed to the it (on any event).

(Note that this is a simplified version of the spec for SFME, so that we can focus on the data structure without spending more time understanding the rest of the components).

Given that specification, what data structure should the event manager use to represent the event IDs and the receivers?

It sounds natural to somehow associate event IDs with receivers, by using a map. But we can't just use `std::map<EventID, Receiver const*>`, because an event ID can have more than one receiver.

We're going to explore two alternative designs and see which one fits most for our event manager:

- a map of vectors: `std::map<EventID, std::vector<EventReceiver const*>>`
- a multimap: `std::multimap<EventID, EventReceiver const*>`

Design 1: A map of vectors

This is probably the most natural design: each event ID can have several receivers, so we map an event ID to a collection of receivers:

```
class EventMediator
{
public:
    void subscribe(EventReceiver const& receiver, EventID eventID);
    void emit(Event const& event) const;
    bool isRegistered(EventReceiver const& receiver) const;

private:
    std::map<EventID, std::vector<EventReceiver const*>>
receiversRegistry_;
};
```

How would the code of the event manager's methods look like with that representation? Let's see the implementation of the three methods: `subscribe`, `emit` and `isRegistered`.

subscribe

The `subscribe` method finds the entry of the map that corresponds to the event ID, and adds a receiver to the corresponding vector or receivers:

```
void EventMediator::subscribe(EventReceiver const& receiver, EventID
eventID)
{
    receiversRegistry_[eventID].push_back(&receiver);
}
```

Simple enough.

emit

The `emit` method picks out the collection of receivers that correspond to the event ID of the event happening, and invoke them all on their `reactTo` method:

```
void EventMediator::emit(Event const& event) const
{
    auto eventID = event.getEventID();
    auto const& receivers = receiversRegistry_[eventID];
    for (auto const& receiver : receivers)
    {
        receiver.reactTo(event);
    }
}
```

Simple too. But this time, the code doesn't compile and triggers the following error:

```
error: no viable overloaded operator[] for type 'const
std::map<EventID, std::vector<const EventReceiver *> >'
```

Behind its rough shell, what this error message is trying to tell us is that we want `emit` to be a `const` method, but `operator[]` is not `const` on the map. Indeed, if the map doesn't have an entry corresponding to the queried event ID, `operator[]` will insert it for us and return a reference to it.

The code to fix the method is less pleasant to the eye:

```
void EventMediator::emit(Event const& event) const
{
    auto eventID = event.getEventID();
    auto receiversEntry = receiversRegistry_.find(eventID);
    if (receiversEntry != end(receiversRegistry_))
    {
        auto const& receivers = receiversEntry->second;
        for (auto const& receiver : receivers)
        {
            receiver->reactTo(event);
        }
    }
}
```

It consists of searching for the event ID, and if we find it in the map then we iterate over the corresponding collection. Note that the nestedness of this piece of code reflects the nestedness of a vector inside a map.

isRegistered

The `isRegistered` method checks if a receiver is registered somewhere in the event manager. Since the map isn't sorted by receivers but only by event IDs (because that's its key), we need to perform a linear search across the whole structure: check the first vector, then the second, and so on:

```
bool EventMediator::isRegistered(EventReceiver const&
searchedReceiver) const
{
    for (auto const& receiversEntry : receiversRegistry_)
    {
        auto const& receievers = receiversEntry.second;
        for (auto const& receiver : receievers)
        {
            if (receiver == &searchedReceiver)
            {
                return true;
            }
        }
    }
    return false;
}
```

Here also, the fact that the data structure is nested leads to a nested code.

The implementation of `subscribe` is fine, but those of `emit` and `isRegistered` could use some simplification, in particular by making them less nested and more straightforward.

Let’s flatten out our data structure by using a multimap instead of a map of vectors.

Design 2: a multimap

A multimap?

What is a multimap, to begin with? It’s like a map, except that a map can only have at most one entry for each key, whereas a multimap can have several entries with equivalent keys.

To illustrate, let’s try to add several entries that have the same key to a `std::map`:

```
auto entries = std::map<int, std::string>{};

entries.insert(std::make_pair(1, "one"));
entries.insert(std::make_pair(1, "uno"));

entries.insert(std::make_pair(2, "two"));
entries.insert(std::make_pair(2, "dos"));

entries.insert(std::make_pair(3, "three"));
entries.insert(std::make_pair(3, "tres"));
```

If we display what the map contains with the following code:

```
for (auto const& entry : entries)
{
    std::cout << entry.first << '-' << entry.second << '\n';
}
```

Here is what the code outputs:

```
1-one
2-two
3-three
```

For each of the keys (1, 2, 3) there is one entry in the map. Now if we replace the map by a multimap:

```
auto entries = std::multimap<int, std::string>{};

…
```

Then the code now outputs:

```
1-one
1-uno
2-two
2-dos
3-three
3-tres
```

There are several entries with equivalent keys.

Replacing the map of vectors with a multimap

In our case, we can use a multimap to associate event IDs with receivers, because some event IDs can be associated with several receivers:

```
class EventMediator
{
public:
    void subscribe(EventReceiver const& receiver, EventID eventID);
    void emit(Event const& event) const;
    bool isRegistered(EventReceiver const& receiver) const;

private:
    std::multimap<EventID, EventReceiver const*> receiversRegistry_;
};
```

Let’s now rewrite our three methods `subscribe`, `emit` and `isRegistered` to see if this new data structure simplifies their implementation.

subscribe

First of all, the standard multimap doesn’t have an `operator[]`: indeed, it is possible that more than one value comes out of a lookup into the multimap. So we have to use the `insert` method:

```
void EventMediator::subscribe(EventReceiver const& receiver, EventID
eventID)
{
    receiversRegistry_.insert(std::make_pair(eventID, &receiver));
}
```

Which is arguably not as elegant as the implementation using `operator[]` that we had with the map of vectors. Let's see how `emit` and `isRegistered` do.

emit

Here is the code for the `emit` function to work with the multimap, we'll go through it line by line:

```
void EventMediator::emit(Event const& event) const
{
    auto eventID = event.getEventID();
    auto receiversEntries = receiversRegistry_.equal_range(eventID);
    for (auto receiverEntry = receiversEntries.first; receiverEntry
!= receiversEntries.second; ++receiverEntry)
    {
        auto const& receiver = receiverEntry->second;
        receiver->reactTo(event);
    }
}
```

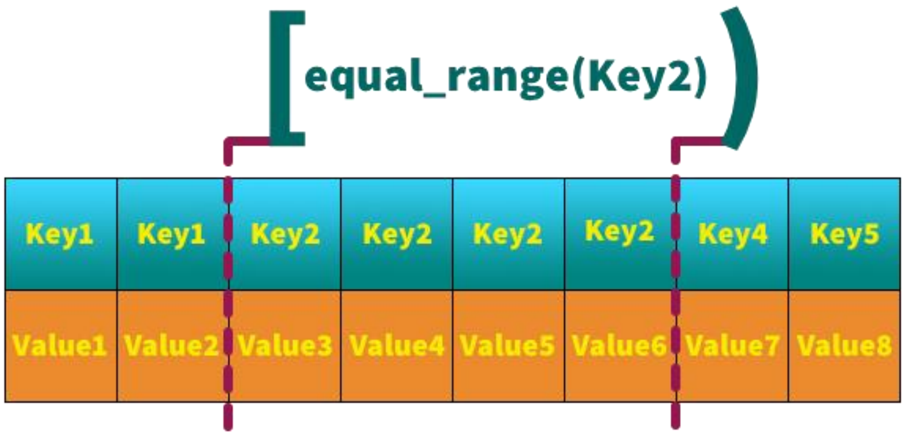
EDIT: as observed by Romain Charbit in the comments section, an `std::for_each` combined with C++14's `auto` in lambdas makes a more concise version:

```
void EventMediator::emit(Event const& event) const
{
    auto eventID = event.getEventID();
    auto receiversEntries = receiversRegistry_.equal_range(eventID);
    std::for_each(receiversEntries.first, receiversEntries.second,
[&event](auto receiverEntry const&)
    {
        auto const& receiver = receiverEntry->second;
        receiver->reactTo(event);
    });
}
```

In case you're not yet familiar with the interface of multimap, here is a line-by-line explanation of the above of code:

```
auto receiversEntries = receiversRegistry_.equal_range(eventID);
```

When we query a multimap for a key, we don't expect to get a value back. Indeed, since the multimap could hold several entries for that key, we get a **range** of entries, which is a slice of the data inside of the multimap:



This sliced could be empty if there wasn't any entry corresponding to the queried key.

```
for (auto receiverEntry = receiversEntries.first; receiverEntry !=
receiversEntries.second; ++receiverEntry)
```

While it makes sense that `equal_range` returns a range, the format of the range returned by the STL here is… not as natural. We would have expected a structure that represents a range, that would have a `begin` and `end` interface, but instead `equal_range` returns a pair of iterators. The first represents the beginning of the range and the second the end.

This integrates badly with the for loop (and with everything else for that matter), hence the complicated above line to simply express "iterate over that range". Anyway, that's an issue with the STL that we had already come across when discussing equal_range to search in STL containers.

```
auto const& receiver = receiverEntry->second;
```

`receiverEntry` is an iterator to an entry in the multimap. The multimap contains `std::pair`s of event IDs and receivers, so to get the receiver we take the `second` of that entry.

```
receiver->reactTo(event);
```

We finally notify the receiver with the event.

Even with the glitch with the interface returned by `equal_range`, this code is overall more straightforward than the `emit` we had with the map of vectors. Indeed, we benefit that the structure is not nested to have code that is not nested either.

isRegistered

As in with the map of vectors, our data structure is still not sorted by receiver. So we have to traverse it linearly and search for a given receiver.

But this time, we only have one layer to traverse, which makes it easier to use an STL algorithm. We could use `std::find_if`, but since we don't need the location of the searched receiver but only whether it is there or not, `std::any_of` will let us go straighter to the point:

```
bool EventMediator::isRegistered(EventReceiver const&
queriedReceiver) const
{
    auto hasQueriedReceiver = [&queriedReceiver](auto const&
receiverEntry){ return receiverEntry.second == &queriedReceiver; };
    return std::any_of(begin(receiversRegistry_),
end(receiversRegistry_), hasQueriedReceiver);
}
```

Or, with a range-based for loop:

```
bool EventMediator::isRegistered(EventReceiver const&
queriedReceiver) const
{
    for (auto const& receiverEntry : receiversRegistry_)
    {
        if (receiverEntry.second == &queriedReceiver)
        {
            return true;
        }
    }
    return false;
}
```

Which are both simpler than the nested version of the map of vectors.

Note that the multimap is probably slightly larger in memory than the map of vectors, because the map of vectors only stores one key for each type of event. But until your memory profiler proved that this extra space is indeed significant (keys are often small, and you may not know the number of values per equivalent key), don't refrain from writing the simplest code.

Map of vectors or multimap?

Even if the map of vectors is maybe more natural to think of at first, the multimap leads to simpler code as soon as we need to **iterate over the data**. This advantage of the multimap come from the fact that it is not a nested structure, contrary to the map of vector.

But does a nested structure always has to lead to nested code? Not necessarily. If you can abstract the fact that it's a nested structure behind a range interface, then the code can behave as if it operated on a flat structure.

One thing that performs this type of abstraction is the `join` range adaptor in range-v3. It can view a vectors of vectors as a flat range that features smart iterators that jumps off a vector to the next and carry out a full traversal of the nested collection as if it was flat.

`join` works on vectors or vectors. But can it work on maps of vectors? There is an additional level of complexity.

Anyway, until we have that sort of components in production, nested structures produce nested code, and flat structures produce flat code. The apple doesn't fall far from the tree.

Thanks to Roman for asking my advice to make code more expressive on the SFME project.

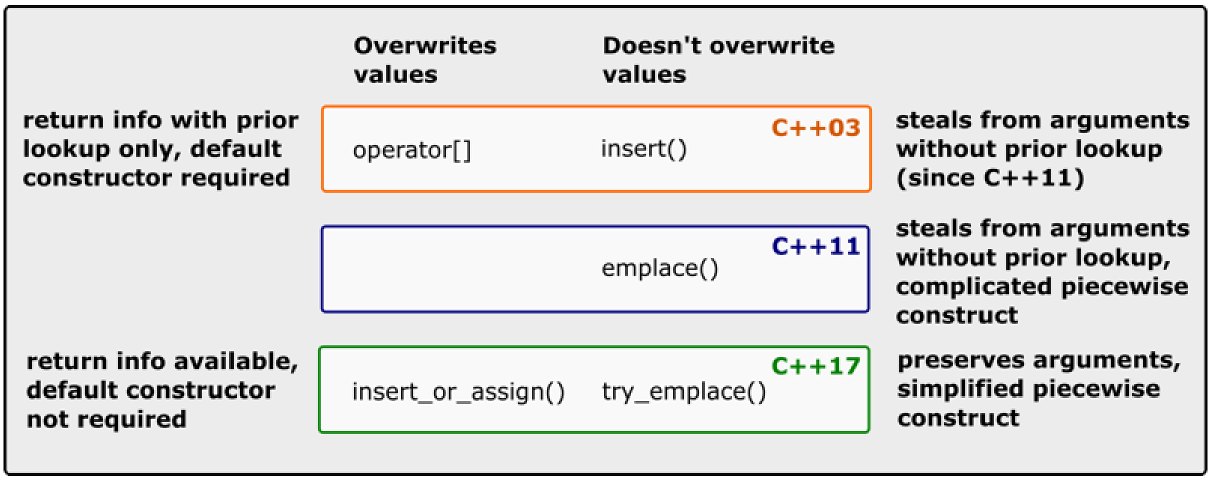
Overview of std::map's Insertion / Emplacement Methods in C++17

Today's guest post is written by @walletfox, one of the hitchhikers in the C++ galaxy, trying to navigate its dark corners by writing articles, creating Mostly Harmless cheat sheets and observing the following rules: "Don't panic! Always carry a cheat sheet next to the towel. So long and thanks for all the fish.".

Interested in writing on Fluent C++ too? Submit your post!

—

TL;DR Go ahead and try out the C++17 emplacement / insertion methods. They offer a tangible improvement in terms of expressiveness and code safety.


Overwrites values
Doesn't overwrite values
return info with prior lookup only, default constructor required
operator[]
insert()
C++03
steals from arguments without prior lookup (since C++11)
emplace()
C++11
steals from arguments without prior lookup, complicated piecewise construct
return info available, default constructor not required
insert_or_assign()
try_emplace()
C++17
preserves arguments, simplified piecewise construct

Here are examples of code using them.

—

C++17 introduced two new insertion / emplacement methods for `std::map`, namely `try_emplace()` and `insert_or_assign()`.

At first sight, this might seem like a cause for concern. Why new insertion methods? Don't we have already plenty of them?

Nevertheless, by studying the problem closer we might come to the conclusion that the introduction of the new methods `try_emplace()` and `insert_or_assign()` makes a convincing case and that they bring us closer to more expressive and safer code.

To understand how we ended up with this pool of insertion / emplacement methods, we are going to use a simple `map<int, std::string>`. Later on, we are going to look at a more complex scenario with `std::map<int, A>` where `A` is a `struct` with two member variables (`std::string`, `int`).

For logging purposes I have provided all the special member functions for the `struct A`. In production, we should apply The Rule of Zero (C++ Core Guidelines, C.20: If you can avoid defining default operations, do) and let the compiler generate the special member functions for us.

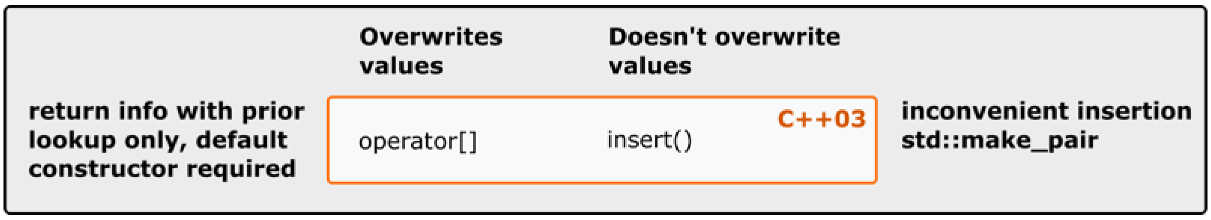
Unless stated otherwise, the code was compiled with gcc 9.0.0. and clang 8.0.0, -O2 (HEAD at the time of writing).

```
struct A
{
    std::string name;
    int age;

    // introduced for logging purposes only
    A(){
        std::cout << "Default ctor. ";
    }

    //explicit
    A(std::string const& s, int x):name(s), age(x){
        std::cout << "Ctor. ";
    }

    A(A const& a):name(a.name),age(a.age){
        std::cout << "Copy ctor. ";
    }

    A(A && a) noexcept
:name(std::move(a.name)),age(std::move(a.age)){
        std::cout << "Move ctor. ";
    }

    A& operator=(A const& a){
        std::cout << "Copy assign. ";
        name = a.name;
        age = a.age;
        return *this;
    }

    A& operator=(A && a) noexcept {
        std::cout << "Move assign. ";
        name = std::move(a.name);
        age = std::move(a.age);
        return *this;
    }

    ~A() noexcept {
        std::cout << "Dtor. ";
    }
};
```

Pre-C++11 days: operator[] and insert()

Before we can discuss how exactly the new C++17 methods `try_emplace()` and `insert_or_assign()` bring improvement, we are going to travel back to pre-C++11 times when all we had was `operator[]` and `insert()`.

The selling point of `operator[]` was its simplicity of use, which unlike `insert()` didn't need to use `std::make_pair()` or other verbose constructs to pass around function arguments.



	Overwrites values	**Doesn't overwrite values**	
return info with prior lookup only, default constructor required	operator[]	insert() **C++03**	**inconvenient insertion std::make_pair**

Fig. 1: The difference between the original insertion methods, C++03

```
// C++03 style
std::map<int, std::string> m;
m[1] = "Ann";

// C++03 style
std::map<int, std::string> m;
m.insert(std::make_pair(1, "Ann"));
```

Convenience aside, what is more important, `operator[]` differs from `insert()` in how it handles a situation when an element with the given key already exists in the map. While `operator[]` simply overwrites the corresponding value, `insert()` doesn't.

```
// C++11 style further ahead
auto m = std::map<int, std::string>{{1, "Ann"}};
m[1] = "Ben";
assert(m.at(1) == "Ben");

auto m = std::map<int, std::string>{{1, "Ann"}};
m.insert({1,"Ben"});
assert(m.at(1) == "Ann");
```

Another important difference lies in the requirements on the `value_type`, namely, `operator[]` requires a `DefaultConstructible value_type`, which means that if we explicitly or implicitly disable the default constructor of the `struct A`, the code won't compile. Notice that, unlike `insert()`, `operator[]` calls different special member functions, i.e. the call to the default constructor is followed by the call to copy/move assignment operator.

```
// Ctor. Default ctor. Move assign. Dtor. Dtor.

auto m = std::map<int, A> {};
m[1] = A("Ann", 63);

// Ctor. Move ctor. Move ctor. Dtor. Dtor. Dtor.
auto m = std::map<int, A> {};
m.insert({1, A("Ann", 63)});
```

Last but not least, these methods differ in the return information they provide. With `operator[]`, we have no way of finding out whether the insertion actually took place, unless we perform a prior lookup. On the other hand, `insert()` returns a `pair<iterator, bool>` that provides us with this information.

Most recently, this has been simplified thanks to structured bindings introduced in C++17.

```
// C++17 structured bindings style
auto[it, ins] = m.insert({2, "Ann"});
```

C++11: move semantics and in-place construction

Further down the road, we got C++11 which introduced move semantics, and both `operator[]` and `insert()`, i.e. the original insertion methods, benefited from this in terms of performance. In addition, C++11 introduced **emplace()** which has the same functionality as `insert()` but additionally, enables **in-place construction**.

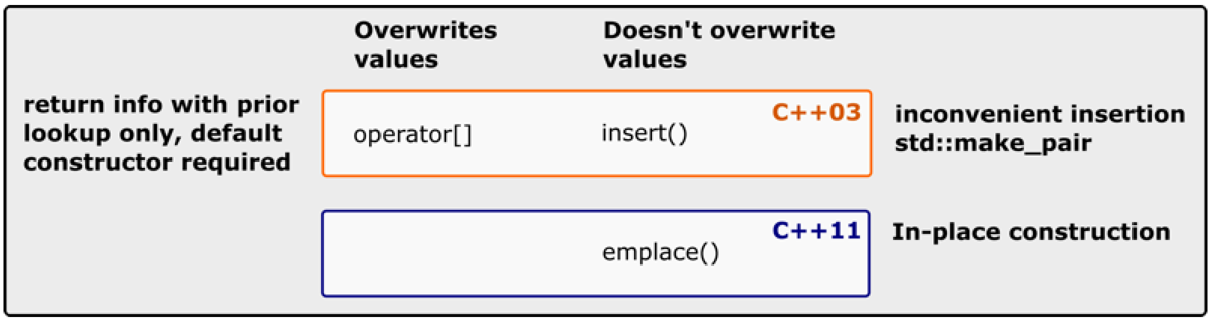

Overwrites values
Doesn't overwrite values
return info with prior lookup only, default constructor required
operator[]
insert()
C++03
inconvenient insertion std::make_pair
C++11
emplace()
In-place construction

Fig. 2: Introduction of `emplace()`, C++11

In-place construction is a technique that bypasses construction and destruction of temporaries by constructing the objects directly in the map. A notable attraction of `emplace`() is that we can do away either with `std::make_pair()` or the extra pair of `{}` that needed to be used with `insert()`. Emplacement is accomplished via perfect forwarding and variadic templates.

The jury is still out on whether `emplace`() should be generally preferred to `insert()`. The potential performance gain is dependent on the types involved and specific library implementations. While Scott Meyers is in favor of `emplace()` (Effective Modern C++, Item 42, what a coincidence!), other C++ experts/guidelines are in favor of `insert()`, most notably Bjarne Stroustrup and Abseil Common Libraries. The reason for that is code safety.

Clang-tidy uses a mixed approach with a general preference for emplacement with the exception of `std::unique_ptr` and `std::shared_ptr` where emplacement could lead to memory leaks:

```
// might leak if allocation fails due to insufficient memory for an
object A
std::map<int, std::unique_ptr<A>> m;
m.emplace(1, std::make_unique<A>("Ann",63));
```

Let's get back to our example and study the effect of different insertion/emplacement constructs. Though this will provide us with some observations, keep in mind that this is a specific example. The types and specific libraries involved are likely to cause differences and it would be counterproductive to make general conclusions. If in doubt, measure.

```
auto m = std::map<int, A> {};

// (1) Ctor. Copy ctor. Move ctor. Dtor. Dtor. Dtor.
m.insert({1, {"Ann", 63}});

// (2) Ctor. Move ctor. Move ctor. Dtor. Dtor. Dtor.
m.insert(std::make_pair(1, A("Ann", 63)));

// (3) Ctor. Move ctor. Move ctor. Dtor. Dtor. Dtor.
m.insert({1, A("Ann", 63)});

// (4) Ctor. Move ctor. Move ctor. Dtor. Dtor. Dtor.
m.emplace(std::make_pair(1, A("Ann", 63))):

// (5) Ctor. Move ctor. Dtor. Dtor.
m.emplace(1, A("Ann", 63)):

// (6) Doesn't compile. That is why try_emplace of C++17 is of
interest
// m.emplace(1, "Ann", 63);

// (7) Ctor. Dtor.
m.emplace(std::piecewise_construct,
std::forward_as_tuple(1),
std::forward_as_tuple("Ann", 63));
```

Now that we have listed some common alternatives, notice that scenario **(1)** resulted in a copy constructor call with both compilers. This is because of copy-list-initialization.

```
// (1) Ctor. Copy ctor. Move ctor. Dtor. Dtor. Dtor.
m.insert({1, {"Ann", 63}});
```

If performance is of concern, we can disable this alternative by marking the multi-argument constructor of `struct A` explicit. This code will then fail to compile:

```
explicit A(std::string const& s, int x):name(s), age(x){
std::cout << "Ctor. ";
}
```

```
// won't compile now, copy-list-initialization prevented
m.insert({1, {"Ann", 63}});
```

It appears that omitting `make_pair()` with `emplace()` in case (5) helped us dispense with one move construction, but we can do even better—this is demonstrated in case (7) where we passed `std::piecewise_construct` and `std::forward_as_tuple` as arguments to `emplace()` resulting in a single constructor and destructor call, completely avoiding intermediate copies and moves!

The verbosity of emplacement with piecewise construct is off-putting, therefore you might appreciate C++17's `try_emplace()` which will do away with the gobbledegook. This is going to be demonstrated in the next section.

For reasons of completeness, I am also listing scenarios where we move from L-values. As you can see, contrary to the previous example, we don't get the same benefit with `emplace()` and `piecewise construct` as before.

```
auto m = std::map<int, A> {};
auto a = A("Ann", 63);

// Ctor. Move ctor. Move ctor. Dtor. Dtor. Dtor.
m.insert(std::make_pair(1, std::move(a)));

// Ctor. Move ctor. Move ctor. Dtor. Dtor. Dtor.
m.insert({1, std::move(a)});

// Ctor. Move ctor. Dtor. Dtor.
m.emplace(1, std::move(a));

// Ctor. Move ctor. Dtor. Dtor.
m.emplace(std::piecewise_construct,
          std::forward_as_tuple(1),
          std::forward_as_tuple(std::move(a)));
```

C++17: try_emplace() and insert_or_assign() as a solution to double lookup

Now we have enough background to understand the rationale behind the introduction of the new methods. `try_emplace()` and `insert_or_assign()` differ in their respective functionalities, but they do have something in common—they are both a solution to a redundant search that had to be performed in pre-C++17 days to provide safety or additional information.

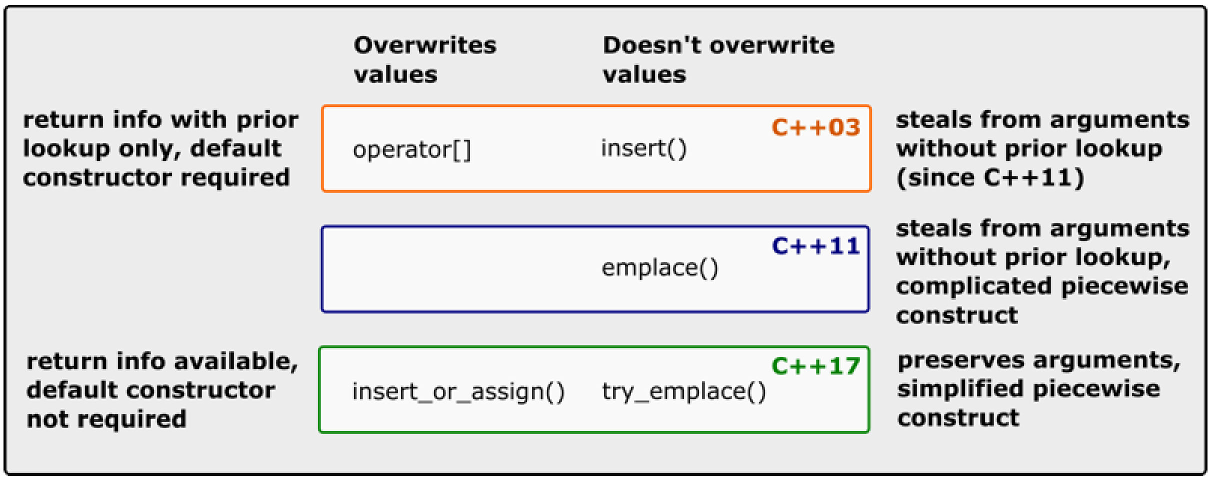

Overwrites values
Doesn't overwrite values
return info with prior lookup only, default constructor required
operator[]
insert()
C++03
steals from arguments without prior lookup (since C++11)
emplace()
C++11
steals from arguments without prior lookup, complicated piecewise construct
return info available, default constructor not required
insert_or_assign()
try_emplace()
C++17
preserves arguments, simplified piecewise construct

Fig. 3 C++17's try_emplace() and insert_or_assign()

try_emplace()

`try_emplace()` is a safer successor of `insert()` or `emplace()`. In line with `insert()` and `emplace()`, `try_emplace()` doesn't modify values for already inserted elements. However, on top of that, it prevents stealing from original arguments that happens both with `insert()` and `emplace()` in case of a failed insertion.

This is demonstrated in the snippet below. An element with key 1 is already in the map, as a result p1 won't be inserted. That doesn't prevent `emplace()` from plundering the pointer p:

```
auto m = std::map<int, std::unique_ptr<A>> {};
m.emplace(1, std::make_unique<A>("Ann",63));

auto p = std::make_unique<A>("John",47);
// p won't be inserted
m.emplace(1, std::move(p));

//but it still might get plundered!!!
assert(p != nullptr); // this will most likely fail
```

In the pre C++17 days this issue could have been only solved with a prior lookup, with `find()`.

```
auto it = m.find(1);
// call emplace only if key doesn’t exist
if (it == m.end()) {
    it = m.emplace(1, std::move(p)).first;
}
assert(p != nullptr);
```

This lookup is no longer necessary. `try_emplace()` makes sure that the argument remains untouched in case it wasn’t inserted:

```
m.try_emplace(1, std::move(p));
// no plundering in case insertion failed
assert(p != nullptr);
```

Although this is the primary purpose of `try_emplace()`, there are some other important advantages. As already mentioned in the previous section, `try_emplace()` simplifies the original `emplace()` that had to use pair’s piecewise constructor:

```
// before C++17
auto m = std::map<int, A> {};
m.emplace(std::piecewise_construct,
          std::forward_as_tuple(1),
          std::forward_as_tuple("Ann", 63));
```

and dispenses with its verbosity in the following way:

```
// C++17
auto m = std::map<int, A> {};
m.try_emplace(1, “Ann”, 63);
```

At first sight, using `try_emplace()` in this way might seem rather user unfriendly due to the nonexistent boundary between the key and the value. However, if used in this way, `try_emplace()` solves another issue of `emplace()`, namely that objects were created even though they weren’t actually used.

Specifically, the map below already contains the key `1` with value `{“Ann”, 63}`, thus a `{“Ben”, 47}` object doesn’t need to be generated, because `emplace()` doesn’t modify values for already existing keys.

```
// std::map m with the original object
auto m = std::map<int, A> {};
m.emplace(1, A("Ann", 63));

// doesn't generate a redundant object
m.try_emplace(1, "Ben", 47);
```

Nonetheless, we shouldn't blindly replace all occurrences of `emplace()` with `try_emplace()` without adjusting the argument list first. **The `try_emplace()` that uses `A`'s constructor below generates a redundant object just like its `emplace()` counterparts:**

```
// Ctor. Dtor. - redundant object
m.try_emplace(1, A("Ben", 47));

// Ctor. Move ctor. Dtor. Dtor.  - redundant object
m.emplace(1, A("Ben", 47));

// Ctor. Dtor. - redundant object
m.emplace(std::piecewise_construct,
std::forward_as_tuple(1),
std::forward_as_tuple("Ben", 47));
```

insert_or_assign()

`insert_or_assign()` is a "smarter" successor of `operator[]`. Just like `operator[]` it modifies values if supplied with a key that is already present in the map. However, unlike `operator[]`, `insert_or_assign()` doesn't require default constructibility of the `value_type`. On top of that, it returns a `pair<iterator, bool>`. The `bool` is `true` when insertion took place and false in case of assignment.

Again, this information was unavailable for `operator[]` without a prior lookup with the help of `find()` as demonstrated below. The map already contains an element with the key `1`, thus this won't be an insertion but an update.

```
auto m = std::map<int, std::unique_ptr<A>> {};
m[1] = std::make_unique<A>("Ann",63);

auto p = std::make_unique<A>("John",47);

auto key = int{1};
auto ins = bool{false};

auto it = m.find(key);
```

```
if(it == m.end()){
    ins = true;
}

m[key] = std::move(p);
assert(ins == false);
```

The code contains a lot of boilerplate that can result both in errors and performance inefficiencies only for the sole purpose of insert or update identification. Luckily, with `insert_or_assign()` we can skip all of it and simply write:

```
auto[it, ins] = m.insert_or_assign(1, std::move(p));
assert(ins == false);
```

Difficulties with inferring from names

At present, it is difficult to conclude whether the new C++17 methods express clearly their intent and functionality. If you have a look at the original proposal, `try_emplace()` is being referred to as `emplace_stable()`, while `insert_or_assign()` is being referred to as `emplace_or_update()`.

At the moment it might seem confusing but with more frequent use we are bound to get it right and hopefully, we will be able to link the new names to the correct functionalities.

Summary

Remember that:

- `insert()`, `emplace()` and `try_emplace()` don't overwrite values for existing keys. On the other hand, `operator[]` and `insert_or_assign()` overwrite them.
- `emplace()` may be susceptible to memory leaks if allocation fails.
- `try_emplace()` doesn't steal from original arguments if insertion fails. This is in contrast to `emplace()` and **`insert()`.**
- `try_emplace()` doesn't generate redundant objects in case insertion didn't take place. This is in contrast to `emplace()`.

- `try_emplace()` offers a simplified piecewise construction. On the other hand, `emplace()` has to use `std::piecewise_construct`, `std::forward_as_tuple`.
- `insert_or_assign()` doesn't require default constructibility. On the other hand, `operator[]` does.
- `insert_or_assign()` returns information on whether insertion or assignment took place. This is in contrast to `operator[]`.

The author is grateful to Jonathan Boccara for hosting, formatting and editing the post and Ricardo Nabinger Sanchez for proofreading.

How to Check If an Inserted Object Was Already in a Map (with Expressive Code)

To insert a new entry into an STL `set` or `map`, or any of their multi- and unordered-equivalents, we use the `insert` method:

```
std::map<int, std::string> myMap = // myMap is initialized with
stuff...

myMap.insert({12, "twelve"});
```

`insert` performs the action of inserting the new entry into the container, if that entry wasn't there already. But `insert` doesn't only perform that action: it also returns two pieces of information about how the insertion went:

- where the new element is now in the map, in the form of an iterator,
- whether the new was actually inserted (it wouldn't be inserted if an equivalent value was already there), in the form of a boolean.

To return those two pieces of information, the `insert` interface of all the associative containers of the STL operate the same way: they return an `std::pair<iterator, bool>`.

This interface makes the code that performs the insertion confusing. Let's see what's wrong with it, and how to improve it.

The problems of the insert interface

Let's focus on the boolean indicating whether the element was inserted, because it has all the problems that the iterator has, plus one more. Say that we want to take a certain action if the element turns out to have been in the map already. There are several ways to write this code. One of them is this:

```
std::pair<std::map<int, std::string>::iterator, bool>
insertionResult = myMap.insert({12, "twelve"});

if (!insertionResult.second)
{
    std::cout << "the element was already in the set.\n";
}
```

This code is horrible for several reasons:

- `std::pair<std::map<int, std::string>::iterator, bool>` is such a big mouthful of code,
- `insertionResult` is not something you'd expect to read in business code,
- the `bool` does not show what it means,
- even if you know the interface of `insert` and that the `bool` depends on whether the element was already there, it's confusing whether it means "successful insertion", or the opposite "the element was already there"
- `insertionResult.second` is meaningless,
- `!insertionResult.second` is meaningless and more complex.

We can mitigate some of its problems by hiding the returned type behind an `auto`, and by naming the `bool` with an explicit name:

```
auto const insertionResult = mySet.insert(12);
auto const wasAlreadyInTheSet = !insertionResult.second;

if (wasAlreadyInTheSet)
{
    std::cout << "the element was already in the set.\n";
}
```

Or, from C++17 on, we can use structured bindings:

```
auto const [position, hasBeenInserted]  = myMap.insert({12,
"twelve"});

if (!hasBeenInserted)
{
    std::cout << "the element was already in the set.\n";
}
```

If you don’t do anything else, at least do this, if you need to check whether the element was already in the container.

I think that this code is OK, but the technical aspects of the `insert` interface are still showing, in particular with the `.second` before C++17, and the risk of having the bool wrong even in C++17. To go further, we can encapsulate it in a function.

A little function that makes the check

A simple way to hide the offending `pair` from the calling code is to wrap the code that gets its `.second` in a function, whose name reveals its intention:

```
template<typename Iterator>
bool wasAlreadyInTheMap(std::pair<Iterator, bool> const&
insertionResult)
{
    return !insertionResult.second;
}
```

Then the calling code looks like this:

```
auto const insertionResult = myMap.insert({12, "twelve"});

if (wasAlreadyInTheMap(insertionResult))
{
    std::cout << "the element was already in the map.\n";
}
```

The ugly `.second` is no longer visible.

The other types of associative containers

Note that this function doesn’t only work for `std::map`. Since all STL associative containers have a similar `insert` interface, it also works for `std::multimap`, `std::unordered_map`, `std::unordered_multimap`, `std::set`, `std::multiset`, `std::unordered_set` and `std::unordered_multiset`.

So the name `wasAlreadyInTheMap` is less generic than what the function can accept. We could rename the function `wasAlreadyInAssociativeContainer`. But even if it's more accurate that `wasAlreadyInTheMap`, the latter looks nicer in calling code.

It is tempting to make a set of overloads for all the different types of STL associative containers:

```
template<typename Key, typename Value>
bool wasAlreadyInTheMap(std::pair<typename std::map<Key,
Value>::iterator, bool> const& insertionResult)
{
    return !insertionResult.second;
}

template<typename Key, typename Value>
bool wasAlreadyInTheMap(std::pair<typename std::multimap<Key,
Value>::iterator, bool> const& insertionResult)
{
    return !insertionResult.second;
}

...
```

But this doesn't work, as this sort of type deduction is not possible. Indeed, the nested type `iterator` is not enough to deduce the container type.

If we want two different names, we can implement two functions differing only by their names, but it doesn't enforce that they will operate with only a `std::set` or `std::map`.

```
template<typename Iterator>
bool wasAlreadyInTheMap(std::pair<Iterator, bool> const&
insertionResult)
{
    return !insertionResult.second;
}

template<typename Iterator>
bool wasAlreadyInTheSet(std::pair<Iterator, bool> const&
insertionResult)
{
    return !insertionResult.second;
}
```

I hope these suggestions will help clarify your code that checks if an elements was inserted in an STL associative container!