What is the difference between unordered_map :: emplace and unordered_map :: insert in C++?

What is the difference between unordered_map :: emplace and unordered_map :: insert in C++ ?

c++ c++11 unordered-map

edited Oct 19 '14 at 4:08
Chris Drew
5,639 1 6 25

asked Oct 19 '14 at 1:31
Harsh M. Shah
53 1 5

3 Taken from a reference: *Careful use of emplace allows the new element to be constructed while avoiding unnecessary copy or move operations. The constructor of the new element (i.e. `std::pair<const Key, T>`) is called with exactly the same arguments as supplied to emplace, forwarded via `std::forward<Args>(args)...` .* – chris Oct 19 '14 at 1:33

emplace creates a new object whereas insert takes an existing object. The parameters are different as emplace takes the constructor arguments. If you don't have an instance already existing to insert, use emplace. – Neil Kirk Oct 19 '14 at 1:34

1 Answer

`unordered_map::insert` copies or moves a key-value pair into the container. It is overloaded to accept reference-to-const or an rvalue reference:

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

template<class P>
std::pair<iterator,bool> insert(P&& value);
```

`unordered_map::emplace` allows you to avoid unnecessary copies or moves by constructing the element in place. It uses perfect forwarding and a variadic template to forward arguments to the constructor of the key-value pair:

```
template<class... Args>
std::pair<iterator,bool> emplace(Args&&... args);
```

But there is a great deal of overlap between the two functions. `emplace` can be used to forward to the copy/move constructor of the key-value pair which allows it to be used just as `insert` would. This means that use of `emplace` doesn't guarantee you will avoid copies or moves. Also the version of `insert` that takes an rvalue-reference is actually templated and accepts any type `P` such that the key-value pair is constructible from `P` .

Scott Meyers says:

> In principle, emplacement functions should sometimes be more efficient than their insertion counterparts, and they should never be less efficient.

(**Edit:** Howard Hinnant ran some experiments that showed sometimes `insert` is faster than `emplace`)

If you definitely do want to copy/move into the container it might be wise to use `insert` because you are more likely to get a compilation error if you pass incorrect arguments. You need to be more careful you are passing the correct arguments to the emplacement functions.

Small example:

```cpp
#include <unordered_map>
#include <iostream>

int main() {
  auto employee1 = std::pair<int, std::string>{1, "John Smith"};

  auto employees = std::unordered_map<int, std::string>{};

  employees.insert(employee1);  // copy insertion
  employees.insert(std::make_pair(2, "Mary Jones"));  // move insertion
  employees.emplace(3, "James Brown");  // construct in-place

  for (const auto& employee : employees)
    std::cout << employee.first << ": " << employee.second << "\n";
}
```

edited May 18 at 11:50

answered Oct 19 '14 at 3:34

Chris Drew

5,639 1 6 25

"Emplacement functions are often more efficient than their insertion counterparts, and they're never less efficient." Howard Hinnant measured something different: htmlpreview.github.io/?https://github.com/HowardHinnant/papers/… See also: groups.google.com/a/isocpp.org/forum/?fromgroups#!searchin/… – dyp Oct 19 '14 at 4:16

@dyp see also not working for me on mobile firefox – Yakk Oct 19 '14 at 9:26

@Yakk Ah, I'm sorry. Hope this works: groups.google.com/a/isocpp.org/d/topic/std-discussion/… `emplace` for associative containers typically creates the element (not just the key) from the arguments even if it's not inserted. So if you already have an element (outside the associative container) and want to copy it, `insert` might be more efficient. – dyp Oct 19 '14 at 12:47

@dyp neat. So for efficiency, we need a partial `pair` construct & type, which we can complete in a second pass. And ditto for `tuple` s probably. – Yakk Oct 19 '14 at 13:05

Alas, no one ever shows an example of a map<T1, T2> where the constructors for T1 and T2 take more than a single argument. I often use things like map<string, classWithTwoParamConstructor>. – jzions Apr 30 at 21:00