How do I add a non-copyable, non-movable object to a std::map or std::unordered_map?

devblogs.microsoft.com/oldnewthing/20231023-00

October 23, 2023

Raymond Chen

Suppose you have a C++ class that is non-copyable and non-movable. This can happen if it has a member which is non-copyable and non-movable, like a `std::mutex`. But how do you put an object of this class into a map?

```
struct weird
{
    int value;
    std::mutex mtx;
};

std::map<int, weird> table;

table.insert({ 1, {} }); // nope
table.insert_or_assign(1, weird{}); // nope
table.emplace({ 1, {} }); // nope
table.emplace(1, weird{}); // nope
table.try_emplace(1, weird{}); // nope
```

The problem with the `insert` method is that it takes a `std::pair<int, weird>` by value, which means that it cannot move the `weird` to its final destination.

The `insert_or_assign`, and `emplace` methods use the parameters to construct a `std::pair<int, weird>` at the final location, but since we passed a `weird{}` object as the second parameter, that would require moving the parameter into its final location.

The `try_emplace` method uses its first parameter as a key and the rest of the parameters to construct the `weird`, so if you pass a brand new `weird`, it's going to use the rvalue or lvalue constructor (as appropriate) to create the `weird` at its final location.

Similar problems exist with `unordered_map`.

One solution is to use `std::piecewise_construct`, (which we learned about some time ago), so that you can specify constructor parameters for each of the halves of the `std::pair`:

```
table.emplace(std::piecewise_construct,
    std::forward_as_tuple(1),
    std::forward_as_tuple());
```

Passing an empty tuple for the second parameter uses the default constructor. If you want to pass constructor parameters for the `weird` object, you can put them inside the second tuple:

```
table.emplace(std::piecewise_construct,
    std::forward_as_tuple(1),
    std::forward_as_tuple("used to construct weird"));
```

The `try_emplace` method uses its first parameter to construct the key and the remaining parameters to construct the mapped type. In our case, we want a default constructor, so we can just say nothing:

```
table.try_emplace(1);
```

If we want to pass constructor parameters for `weird`, we can pass them after the key:

```
table.try_emplace(1, "parameters for constructing weird");
```

Bonus chatter: There are versions of these insertion and emplacement methods which take hints as the first parameter, ahead of the value or key. We learned about hints some time ago.