The std::map subscript operator is a convenience, but a potentially dangerous one

devblogs.microsoft.com/oldnewthing/20190227-00

February 27, 2019

Raymond Chen

The `std::map<K, V>` class is an associative container of key/value pairs. It also provides a handy subscript operator `[]` that lets you access the map as if it were an array.

That operator is a convenience, but a potentially dangerous one.

The problem is that the `[]` operator is trying to serve two masters. You might use it to read a value from the map:

```
std::map<int, T> m;

auto value = m[2]; // retrieve the item whose key is 2
```

Or you might use it to write to the map.

```
m[2] = t; // create or update the item whose key is 2
```

The `[]` operator doesn't know whether you're going to read from or write to the result, so it has to come up with some sort of compromise. And sometimes the result of a compromise is something both sides dislike.

- If the key is found in the map, then return a reference to the existing value. You can read that value or overwrite it.
- If the key is not found in the map, then create a new entry in the map consisting of the key (moved, if movable) and a default-constructed value, then return a reference to that freshly-created value. You can read that value or overwrite it.

This is different from how the `[]` operator works in C#, JavaScript, Java, Perl, PHP, Python, Ruby, pretty much every other programming language out there. In all those other languages, reading a value out of an associative array is a non-mutating operation. Trying to fetch an element whose key is `2` does not cause such an element to be created!

But that's what happens when you read from a `std::map` using the `[]` operator.

Also, in those other languages, setting a new value in the associative array does not first create a throwaway value that gets immediately overwritten.

But that’s what happens when you create a new key/value pair in a `std::map` using the `[]` operator.

Synthesizing a value in the `[]` operator means that you can accidentally fill your `std::map` with garbage entries when you really were just trying to see if an element was there.

```
std::map<int, std::unique_ptr<T>> m;

void RecolorizeIfPresent(int i)
{
  if (m[i]) { m[i]->Recolorize(); }
}
```

If you call `RecolorizeIfPresent` with an integer that is not in the associative array, the associative array will create an empty `std::unique_ptr<T>`, add it to the associative array, and then return it to you. Your `if` then converts that `std::unique_ptr<T>` to a `bool`, which says whether the `std::unique_ptr<T>` is managing a `T`, which in the case of a default-constructed `std::unique_ptr<T>`, will be “no”.

The result is that your map gets clogged with empty `std::unique_ptr<T>` objects, one for each key you probed. If you aren’t expecting this behavior, you just created a memory leak: You didn’t expect the probe to create an entry in the map, so you aren’t going to have any code to clean out those empty entries. Those empty entries will just keep accumulating, slowly consuming more and more memory.

Meanwhile, the code that is trying to add an entry to the map is also not too happy.

```
std::map<int, T> m;

m[2] = T(constructor_argument1, constructor_argument2);
```

It looks like this code creates a `T` object and puts it into the map under the key `2`. But actually, this code creates *two* `T` objects. It creates a default one when you say `m[2]` and there is no entry for `2`, and then it creates a second one when you construct one with `T(...)`. Finally, the assignment operator causes the first one to be overwritten by the second one, and then the second one is destructed.

This can be quite expensive if `T` has a complex constructor, or if its constructor has observable side effects.

```
class Screenshot
{
public:
  // The default constructor captures the entire screen.
  Screenshot();

  // Or you can specify a portion of the screen.
  Screenshot(Rectangle const& rect);

  ...
};

std::map<Timestamp, Screenshot> screenshots;

void CaptureScreenRectangle(Rectangle const& rect)
{
 m[now()] = Screenshot(rect);
}
```

Every time you call `CaptureScreenRectangle`, you actually take a screen shot of the entire screen (which will probably be slow and allocate a lot of memory), and then take a screen shot of the desired rectangle (which could very well be tiny), and then overwrite the giant screen shot with the small one.

Note also that if the value type is not default-constructible, then the `[]` operator won't work at all!

```
struct Nope
{
public:
  Nope(int); // no default constructor
};

std::map<int, Nope> m;

m[2] = Nope(2); // does not compile
```

My personal recommendation is to avoid the `[]` operator. If you want to use an entry if it exists, then use `map::find()`.

```
auto item = m.find(2);
if (item != m.end()) {
 DoSomethingWith(*item);
}
```

If you expect the entry to exist, then use `map::at()`.

```
auto& value = m.at(2); // throws if not found
```

If you want to create a brand new entry, but leave any existing entry intact, then use `map::insert` or `map::emplace` or `map::try_emplace`.

```
auto [item, inserted] = m.insert({ now(), Screenshot(rect) });

auto [item, inserted] = m.emplace(now(), rect);

// try_emplace doesn't even construct the Screenshot if an
// entry already exists.
auto [item, created] = m.try_emplace(now(), rect);
```

If you want to create a new entry or overwrite any existing entry, then use `map::insert_or_assign`.

```
auto [item, inserted] = m.insert_or_assign(now(), Screenshot(rect));
```

Just don't use `[]`.

Bonus reading: Overview of std::maps Insertion / Emplacement Methods in C++17.

Bonus chatter: There have been various ideas for fixing this problem with `[]`, but none of them work.

Bonus viewing: Louis Brandy calls this "The greatest C++ newbie trap."

Raymond Chen

Follow