Solving the puzzle of trying to put an object into a std::optional

devblogs.microsoft.com/oldnewthing/20241114-00

November 14, 2024

Last time, we investigated the puzzle of why the compiler wouldn't let us put an object into a `std::optional`. It came down to the fact that the object is not copy-constructible, move-constructible, copy-assignable, or move-assignable, so there's no way to put the temporary object into the `std::optional`.

What we have to do is *construct* the object in place inside the `std::optional`. And the C++ standard library term for "construct an object inside a container" is "emplace".

```
struct Doodad
{
    Doodad();
    ~Doodad();
    std::unique_ptr<DoodadStuff> m_stuff;
};

struct Widget
{
    std::optional<Doodad> m_doodad;

    Widget()
    {
        if (doodads_enabled()) {
            m_doodad.emplace();
        }
    }
};
```

The parameters to `emplace` are whatever parameters you would have passed to the `Doodad` constructor. In our case, we wanted the default constructor, so that means that we pass nothing to `emplace()`.

I may as well take this time to review the various options for placing a value into a `std::optional`, because they are subtly different. For the purpose of this discussion, the `T` object being held inside a `std::optional<T>` will be called its "value".

	Previous state of `optional<T>`	
	Empty	**Not empty**
`o.reset();`	Nothing happens	Destruct the existing `T`
`o.emplace(args...);`	Construct a `T` from `args...`	Destruct the existing `T` and construct a new `T` from `args...` same as `o.reset();` `o.emplace(args...);`
`o = v;`	Construct a `T` from `v` same as `o.emplace(v)`	Assign `v` to the existing value same as `*o = v;`
`o = std::nullopt;`	Nothing happens	Destruct the existing `T`

Note that the `o = v;` might construct the object, or it might assign the object, depending on the prior state of the `std::optional`. That's why the requirements for the assignment operator require both constructibility and assignability from the right hand side. If you already know whether the object is empty or nonempty, you avoid the compiler having to generate code for both possibilities by going straight to `emplace()` method (if you know that it is empty), or going straight to `T`'s assignment operator `*o = v;` (if you know that it is nonempty). Note, though, that the penalty for guessing wrong varies depending on the path you take.

	Previous state of `optional<T>`	
	Empty	**Not empty**
`o.emplace(v);`	Construct a `T` from `args...`	Destruct the existing `T` and construct a new `T` from `v...`
`o.value() = v;`	`std::bad_optional_access` exception	Assign `v` to the existing value
`*o = v;`	Undefined behavior	Assign `v` to the existing value

If you try to emplace thinking that the optional is empty, but it is in fact nonempty, then instead of assigning the value, you destruct the old value and then construct a new one. This is a subtle difference, but it is significant because it runs the object's destructor and then re-runs the object's constructor.

On the other hand, if you use `o.value() = v;` to assign a value when the optional is empty, you get a runtime exception. And even worse, if you use `*o = v;` to assign a value when the optional is empty, you get undefined behavior, and that's super-bad.

But wait, what if your type is not copyable, movable, *or* constructible? For example, maybe instead of a constructor, it has a factory method. How can you put one of these objects into a `std::optional`? We'll look at that next time.

Bonus chatter: In the case where you `emplace()` into a nonempty optional, the old value is destructed, and the new value is constructed. If the construction of the new value throws an exception, then the optional stays empty. This is another subtle difference from using the assignment operator, because a failed assignment does *not* destruct the optional's value.

Bonus bonus chatter: You might think you can move a value out of an optional by doing

```
auto v = std::move(v.value());
```

but while this does move the value's contents, the optional remains nonempty, with a moved-from value inside it. Even move-constructing an optional from an optional *does not empty the source*.

```
std::optional<T> p = std::move(o);
// o is nonempty and contains a moved-from value
```

If you want to empty the optional, you can exchange it.

```
std::optional<T> p = std::exchange(o, std::nullopt);
```

You can equivalently write

```
std::optional<T> p = std::exchange(o, {});
```

but for some reason, msvc and gcc fail to optimize out the temporary empty `std::optional<T>{}`, so stick with `std::nullopt`.

Bonus bonus bonus chatter: If you want to construct a `std::optional` with an object already inside it, you can use `in_place` with constructor arguments.

```
// constructs the Doodad as if by Doodad(x, y, z)
std::optional<Doodad> o(std::in_place, x, y, z);
```