How can I explicitly specialize a templated C++ constructor, follow-up notes

devblogs.microsoft.com/oldnewthing/20241025-00

October 25, 2024

A brief follow-up on How can I explicitly specialize a templated C++ constructor?, which pedantically should probably have been titled something like “How can I explicitly *instantiate* a templated C++ constructor?” but what’s done is done.

Our solution was to use the `in_place_type` type-holder class to specify what type of object the object manager should contain:

```
struct ObjectManager
{
    // Concrete should derive from CommonBase
    template<typename Concrete, typename...Args>
    ObjectManager(int reason,
        std::in_place_type_t<Concrete>,
        Args&&...args) :
        m_base(std::make_unique<Concrete>(
                *this, std::forward<Args>(args)...))
    {
        m_base->initialize(reason);
    }

    std::unique_ptr<CommonBase> m_base;
};
```

We could also apply the “give it a name” principle to the problem and offer a factory method, which is easier to instantiate with an explicit type.

```
struct ObjectManager
{
    template<typename Concrete, typename...Args>
    static ObjectManager make(int reason, Args&&...args)
    {
        return ObjectManager(reason,
            std::in_place_type<Concrete>
            std::forward<Args>(args)...));
    }

    ⟦ ... as before ... ⟧
};

// Example usage:
auto manager = ObjectManager::make<Derived>(9, 42);
```

Note that our solution still uses a type tag parameter (in this case, `in_place_type`). This is unavoidable because the `ObjectManager` constructor uses a reference to itself. This is a Catch-22 we ran into when we tried to construct nodes of a hand-made linked list. Consider this alternative:

```
struct ObjectManager
{
    template<typename Concrete, typename...Args>
    static ObjectManager make(int reason, Args&&...args)
    {
        // Code in italics is wrong
        ObjectManager manager(reason,
            std::make_unique<Concrete>(
                manager, std::forward<Args>(args)...));
        return manager;
    }

    template<typename Trait, typename...Args>
    ObjectManager(int reason, std::unique_ptr<CommonBase> base) :
        m_base(std::move(base))
    {
        m_base->initialize(reason);
    }

    std::unique_ptr<CommonBase> m_base;
};
```

This version creates an `ObjectManager` that handed out a reference to itself (`*this`), and then moves that `ObjectManager` to the return value, thereby changing its address and invalidating the reference.

The above code is eligible for named return value optimization (NRVO), in which case the `manager` object can be constructed directly in the return value slot. However, NRVO is not a mandatory optimization, so a compiler is permitted to construct the `manager` separately and

then move it[1] to the return value slot when the function exits.

Return value optimization (RVO, formally called *copy elision*) is guaranteed in the case where you return a freshly-constructed object, which is why our initial version takes the form `return ObjectManager(...)`.

[1] Even though you didn't explicitly write `std::move(manager)`, the compiler is required to move it. In fact, if you write `return std::move(manager);`, you have inadvertently defeated the `std::move` optimization!