Making sure that people use make_unique and make_shared to make your object

devblogs.microsoft.com/oldnewthing/20220721-00

July 21, 2022

Raymond Chen

Normally, the way you prevent people from constructor your object in an objectionable way is by making the constructor private and offering a factory method. That way, the only way to create the object is through the factory.

```
class Widget
{
public:
    template<typename...Args>
    static auto Create(Args&&... args)
    { return Widget(std::forward<Args>(args)...); }

private:
    Widget();
    Widget(int);
    Widget(int, int);

    // Deny copy construction
    Widget(Widget const&) = delete;
};
```

However, this trick doesn't work with `make_unique` `make_shared` because they require that the construct be public, so that they can create the object that is being managed.

For `make_unique`, you can work around it by `new`'ing the object yourself and putting it directly into a `unique_ptr`:

```
class Widget
{
public:
    template<typename...Args>
    static auto Create(Args&&... args)
    { return std::unique_ptr<Widget>
        (std::forward<Args>(args)...); }
private:
    Widget();
    Widget(int);
    Widget(int, int);

    // Deny copy construction
    Widget(Widget const&) = delete;
};
```

You can also do this to bypass `make_shared`, but it's not quite the same because you lose the special optimization in `make_shared` that allocates the object and its control block inside the same memory allocation.

Since we generally like the "combined allocation" optimization, we are forced to make the constructor public. To avoid unwanted use of the constructor, we can make it impossible to invoke, using a trick we learned some time ago: Require an additional parameter of an inaccessible marker type.

```
class Widget : public std::enable_shared_from_this<Widget>
{
private:
    struct secret { explicit secret() = default; };
public:
    template<typename...Args>
    static auto Create(Args&&... args)
    { return std::make_shared<Widget>
        (secret{}, std::forward<Args>(args)...); }

    // public but unusable from outside the class
    Widget(secret);
    Widget(secret, int);
    Widget(secret, int, int);

    // Deny copy construction
    Widget(Widget const&) = delete;
};
```

The public constructors take an instance of a private type called `secret`. The constructor of this private type is explicit so users can't use the `{}` trick to construct the object without naming it.

We can capture this pattern in a helper class:

```
template<typename T>
struct require_make_shared :
    public std::enable_shared_from_this<T>
{
protected:
    struct use_the_create_method {
        explicit use_the_create_method() = default;
    };

public:
    template<typename...Args>
    static auto create(Args&&... args)
    {
        return std::make_shared<T>
            (use_the_create_method{},
             std::forward<Args>(args)...);
    }

    // Deny copy construction
    require_make_shared(require_make_shared const&) = delete;
};

class Widget : public require_make_shared<Widget>
{
public:
    Widget(use_the_create_method);
    Widget(use_the_create_method, int);
    Widget(use_the_create_method, int, int);
};

void test()
{
    auto v1 = Widget::create(); // okay
    auto v2 = std::make_shared<Widget>(); // nope
    auto v3 = std::make_unique<Widget>(); // nope
    Widget v4; // nope
}
```

We name the marker type `use_the_create_method` as another example of compiler error message metaprogramming: If somebody tries to create one of these things directly, they get an error message that includes the phrase "`use_the_create_method`", and that might clue them in that they need to use the `create()` method.

Notice that if you forget to derive publicly from `require_make_shared`, then you can't even `create` one!

```
class Widget : /* private */ require_make_shared<Widget>
{
public:
    Widget(use_the_create_method);
    Widget(use_the_create_method, int);
    Widget(use_the_create_method, int, int);
};

void test()
{
    auto v1 = Widget::create(); // nope
}
```

This is probably good enough to protect against simple mistakes. To protect against fancier mistakes, you could add a static assertion:

```
template<typename T>
struct require_make_shared :
    public std::enable_shared_from_this<T>
{
protected:
    struct use_the_create_method {
        explicit use_the_create_method() = default;
    };

public:
    template<typename...Args>
    static auto create(Args&&... args)
    {
        static_assert(std::is_convertible_v<
            T*, require_make_shared*>,
            "Must derive publicly from require_make_shared");
        return std::make_shared<T>
            (use_the_create_method{},
             std::forward<Args>(args)...);
    }

    // Deny copy construction
    require_make_shared(require_make_shared const&) = delete;
};

class Widget : /* private */ require_make_shared<Widget>
{
public:
    auto create()
    { return require_make_shared::create(); } // assertion failure
    Widget(use_the_create_method);
    Widget(use_the_create_method, int);
    Widget(use_the_create_method, int, int);
};
```

Raymond Chen

Follow