How can I write a C++ class that iterates over its base classes?

devblogs.microsoft.com/oldnewthing/20210114-00

January 14, 2021

Raymond Chen

Suppose you have a class with multiple base classes, and you want to invoke a method on all of the base classes.

For example, say we have `Pillow` and `Radio` classes:

```
class Pillow
{
public:
    int price();
    int weight();
    void refurbish(int level);
};

class Radio
{
public:
    int price();
    int weight();
    void refurbish(int level);
};
```

And you want to create a `PillowRadio`, which is a combination pillow and radio. It is basically a pillow and a radio glued together. Okay, this is kind of ridiculous because there is no such thing as a pillow-radio,[1] but let's go along with it.

We would like the PillowRadio class to go something like this, assuming there were some way to iterate over the base classes, for which I have made up some hypothetical syntax.

```
class PillowRadio : public Pillow, public Radio
{
public:
    int price()
    {
        int total = 0;
        for (typename T : base_classes_of(this)) {
            total += T::price();
        }
        return total + 10; /* extra 10 for packaging */
    }

    int weight()
    {
        int total = 0;
        for (typename T : base_classes_of(this)) {
            total += T::weight();
        }
        return total + 5; /* extra 5 for packaging */
    }

    void refurbish(int level)
    {
        for (typename T : base_classes_of(this)) {
            T::refurbish(level);
        }
    }
};
```

The point is that you may have cases where you want to iterate over your base classes and aggregate the results.

So how do you do this?

C++ doesn't provide this degree of reflection but you can simulate it by introducing a helper class.

```
template<typename... bases>
struct Aggregator : bases...
{
    int price()
    {
        return (0 + ... + bases::price());
    }

    int weight()
    {
        return (0 + ... + bases::weight());
    }

    void refurbish(int level)
    {
        (bases::refurbish(level), ...);
    }
};

class PillowRadio : Aggregator<Pillow, Radio>
{
public:
    int price()
    {
        return Aggregator::price() + 10; /* extra 10 for packaging */
    }

    int weight()
    {
        return Aggregator::weight() + 5; /* extra 5 for packaging */
    }

    /* inherit refurbish from Aggregator */
};
```

How does this work?

The `Aggregator` class is given a list of base classes, and it dutifully derives from them. So that solves the first problem: Deriving from an `Aggregator` causes you to derive from all of the specified base classes.

The methods on `Aggregator` use fold expressions which iterate over the template type parameters and combine the results in some way.

For the case of `refurbish`, we don't actually have any results to combine; we just want to invoke `refurbish` on each base class, so we use the comma operator to throw the results away after invoking each method. Fortunately, `refurbish` returns `void`, so we don't have to worry about somebody doing a sneaky overload of the comma operator.

Of course, this `Aggregator` is tightly coupled to the methods of its base classes. Maybe we can generalize it.

```
template<typename... bases>
struct Aggregator : bases...
{
    template<typename Visitor>
    void for_each_base(Visitor&& visitor)
    {
        (void(visitor(static_cast<bases&>(*this))), ...);
    }
};
```

The `for_each_base` method takes a visitor functor and calls it once for each base class. We cast the result to `void` so that we can safely use the comma fold operator to throw the results away after each call of the visitor.

Now we can implement the aggregator methods for our `PillowRadio` class.

```
class PillowRadio : Aggregator<Pillow, Radio>
{
public:
    int price()
    {
        int total = 10; /* extra 10 for packaging */
        for_each_base([&](auto&& base) { total += base.price(); });
        return total;
    }

    int weight()
    {
        int total = 5; /* extra 5 for packaging */
        for_each_base([&](auto&& base) { total += base.weight(); });
        return total;
    }

    void refurbish(int level)
    {
        for_each_base([&](auto&& base) { base.refurbish(level); });
    }
};
```

Okay, but what about static members?

Since function parameters cannot be types, we have to encode the type in the parameter somehow, say by passing a suitably-cast null pointer.

```
template<typename... bases>
struct Aggregator : bases...
{
    template<typename Visitor>
    void for_each_base(Visitor&& visitor)
    {
        (void(visitor(static_cast<bases&>(*this))), ...);
    }

    template<typename Visitor>
    static void static_for_each_base(Visitor&& visitor)
    {
        (void(visitor(static_cast<bases*>(nullptr))), ...);
    }
};
```

This time, the lambda gets a null pointer of the appropriate type. You can then access static members via that strongly-typed null pointer.

```
class Pillow
{
public:
    static int list_price();
};

class Radio
{
public:
    static int list_price();
};

class PillowRadio : Aggregator<Pillow, Radio>
{
public:
    static int list_price()
    {
        int total = 10; /* extra 10 for packaging */
        static_for_each_base([&](auto* base) {
            using Base = std::decay_t<decltype(*base)>;
            total += Base::list_price();
        });
        return total;
    }
};
```

Even though the visitor is given a pointer, that pointer is always null. It is useful only for its type information, not for its value.

It is somewhat unclear whether it is permissible to access static members via a strongly-typed null pointer, so this alternative seems somewhat risky:

```
        // dereferencing null pointer to access static member - unclear legality
        static_for_each_base([&](auto* base) { total += base->list_price(); });
```

C++20 adds the ability to name the deduced template types of a lambda, so this becomes slightly less awkward:

```
        static_for_each_base([&]<typename Base>(Base*) { total += Base::list_price();
});
```

You might want the static and nonstatic versions of `for_each_base` to agree on the type of the parameter passed to the visitor, in which case you can have the nonstatic version also pass a pointer:

```
template<typename... bases>
struct Aggregator : bases...
{
    template<typename Visitor>
    void for_each_base(Visitor&& visitor)
    {
        (void(visitor(static_cast<bases*>(this))), ...);
    }

    template<typename Visitor>
    static void static_for_each_base(Visitor&& visitor)
    {
        (void(visitor(static_cast<bases*>(nullptr))), ...);
    }
};

class PillowRadio : Aggregator<Pillow, Radio>
{
public:
    int price()
    {
        int total = 10; /* extra 10 for packaging */
        for_each_base([&](auto* base) { total += base->price(); });
        return total;
    }

    static int list_price()
    {
        int total = 10; /* extra 10 for packaging */
        static_for_each_base([&](auto* base) {
            using Base = std::decay_t<decltype(*base)>;
            total += Base::list_price();
        });
        return total;
    }
};
```

This aligns the two versions, but it may also make it easier to mistakenly move code from the non-static version to static version without realizing that the meaning of the pointer has changed. I'll let you decide which is better.

A final consolidation could be merging the instance and static versions by taking an explicit starting point for the aggregator, either null or non-null.

```
template<typename... bases>
struct Aggregator : bases...
{
    template<typename Visitor>
    static void for_each_base(Aggregator* self, Visitor&& visitor)
    {
        (void(visitor(static_cast<bases*>(self))), ...);
    }
};

class PillowRadio : Aggregator<Pillow, Radio>
{
public:
    int price()
    {
        int total = 10; /* extra 10 for packaging */
        for_each_base(this, [&](auto* base) { total += base->price(); });
        return total;
    }

    static int list_price()
    {
        int total = 10; /* extra 10 for packaging */
        // C++20: [&]<typename Base>(Base*) {
        for_each_base(nullptr, [&](auto* base) {
            using Base = std::decay_t<decltype(*base)>;
            total += Base::list_price();
        });
        return total;
    }
};
```

[1] Though fans of a Swedish children's television show from 2004 may remember an episode that involved such a contraption, with the obvious name *kudderadio*. (Sorry, I couldn't find a link to the *kudderadio* episode.)

Raymond Chen

Follow