Perfect forwarding forwards objects, not braced things that are trying to become objects

devblogs.microsoft.com/oldnewthing/20230727-00

July 27, 2023

Raymond Chen

In C++, perfect forwarding is the act of passing a function's parameters to another function while preserving its reference category. It is commonly used by wrapper methods that want to pass their parameters through to another function, often a constructor.

```
template<typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args)
{
    return std::unique_ptr<T>(
        new T(std::forward<Args>(args)...));
}
```

The `make_unique` function takes its parameters, forwards them to the `T` constructor, and then puts the pointer to the newly-constructed `T` inside a `unique_ptr`. Those parameters are forwarded perfectly into the constructor: If the original parameters were rvalue reference, then the constructor receives rvalue reference. If the original parameters were lvalue references, then the constructor receives lvalue reference.

But the catch is that it can forward only *objects*. It can't forward "braced things that are trying to become objects".

```
struct Point
{
    int x, y;
};

struct Segment
{
    Segment(Point p1, Point p2);
};

void test()
{
    // This works
    Segment s({ 1, 1 }, { 2, 2 });

    // This doesn't
    auto p = std::make_unique<Segment>(
        { 1, 1 }, { 2, 2 });
}
```

The constructor of `Segment` says that it wants two `Point`s, and if you pass some stuff enclosed in braces, the compiler knows, “Well, I should make a `Point` out of this.”

On the other hand, the parameters to `make_unique` are generic and are eventually forwarded to the constructor. But at the time the compiler sees the call to `std::make_unique`, all it sees is a bunch of stuff inside curly braces.

And a bunch of stuff inside curly braces is not an object.[1] It’s a thing that can help initialize an object, but it’s not itself an object. It has no type, and it cannot be stored in a variable or parameter. I mean, we know by our powers of clairvoyance that the stuff inside curly braces is eventually going to be used to construct a `Point`, and that `make_unique` doesn’t ever do anything that needs to know its type; it doesn’t do `decltype(arg1)` or `auto arg1_copy = arg1;`.

We know that because we read ahead and saw what the `make_unique` function does with the parameters. But the compiler doesn’t look that far ahead.[2]

Perfect forwarding is not perfect.

You can work around this by providing your own wrappers that accept the same parameters as the constructors. Since these parameters have types, the compiler knows what “things inside braces” are supposed to be.

```
struct Segment
{
    Segment(Point p1, Point p2);

    static std::unique_ptr<Segment> make_unique(
        Point p1, Point p2)
    {
        return std::make_unique<Segment>(p1, p2);
    }
};
```

Now, we are kind of cheating here because we are copying the `Point` objects all over the place. We get away with it because a `Point` is a very lightweight class. On the other hand, if it were more expensive (or even impossible) to copy a `Point`, we could accept a reference and forward it. This time, we can use a forwarding reference, but give the template types a *hint*:

```
struct Segment
{
    Segment(Point p1, Point p2);

    template<typename Arg1 = Point,
             typename Arg2 = Point>
    static std::unique_ptr<Segment> make_unique(
        Arg1&& p1, Arg2&& p2)
    {
        return std::make_unique<Segment>(
            std::forward<Arg1>(p1),
            std::forward<Arg2>(p2));
    }
};
```

This accepts anything for the two parameters and forwards them perfectly to `std::make_unique`. Providing defaults for the `Arg1` and `Arg2` template type parameters tells the compiler, "And if you can't figure out what type it is, try this." That way, if the caller passes a bunch of junk inside curly braces, the compiler will try to use it to construct a `Point`.

[1] Okay, it might become an `initializer_list`, but that comes later.

[2] And in the case where the function being called is visible only as a prototype, the compiler couldn't look into the body even if it wanted to.