If you’re not keeping the parameter, then you still want to have separate T const& and T&& overloads

devblogs.microsoft.com/oldnewthing/20200220-00

February 20, 2020

Raymond Chen

Last time, I noted that if you plan on keeping the parameter anyway, then there’s no need to have separate `T const&` and `T&&` overloads. However, the converse also applies: If you’re not keeping the parameter, then you still want to have separate `T const&` and `T&&` overloads.

To recap, we started with a class like this:

```
class Widget
{
public:
  void SetValues(std::vector<int> const& values)
  {
    m_values = values;
  }

  void SetValues(std::vector<int>&& values)
  {
    m_values = std::move(values);
  }
private:
  std::vector<int> m_values;
};
```

We were able to simplify this to

```
class Widget
{
public:
  void SetValues(std::vector<int> values)
  {
    m_values = std::move(values);
  }

private:
  std::vector<int> m_values;
};
```

because we are going to keep the parameter either way. (The old way resulted in either a copy or a move. The new way produces either a copy+move or a move. The expectation is that a single move is relatively inexpensive.)

However, the simplification doesn't apply if we are not the ones consuming the value.

```
Widget CreateWidgetWithValues(std::vector<int> values)
{
  Widget widget;
  widget.SetValues(std::move(values));
  return widget;
}
```

In this case, we are moving the `values` onward to the `SetValues` method, who is the final consumer. Writing the method this way generates an extra move constructor, because we have to move the value from our inbound parameter into the outbound parameter to `SetValues`. We also incur an extra destruction of our now-empty inbound parameter. If the parameter is passed through multiple layers, each layer adds an extra move constructor and destruction.

Since we are not the final consumer, we should forward the parameter.

```
template<typename Values>
Widget CreateWidgetWithValues(Values&& values)
{
  Widget widget;
  widget.SetValues(std::forward<Values>(values));
  return widget;
}
```

Unfortunately, this causes us to break existing code, since you cannot forward uniform initialization.

```
// doesn't work any more
CreateWidgetWithValues({ range.begin(), range.end() });
```

We end up returning to the overload.

```
Widget CreateWidgetWithValues(const std::vector<int>& values)
{
  Widget widget;
  widget.SetValues(values);
  return widget;
}

Widget CreateWidgetWithValues(std::vector<int>&& values)
{
  Widget widget;
  widget.SetValues(std::move(values));
  return widget;
}
```

I’m not too happy with this, though. Maybe there’s an easier way. Let me know.

Bonus chatter: The Microsoft compiler makes the function responsible for destructing its inbound parameters, in which case the code to destruct the `std::vector<int>` is part of the consuming function and is therefore shared. gcc and clang make it the caller’s responsibility, so the destruction of the parameter is repeated at each call site.

Raymond Chen

Follow