If you plan on keeping the parameter anyway, then there's no need to have separate T const& and T&& overloads

devblogs.microsoft.com/oldnewthing/20200219-00

February 19, 2020

Raymond Chen

Suppose you have a function that takes a `std::vector` and it wants to take ownership of the vector. You might decide to write two overloads:

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

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

If the caller passes a read-only lvalue reference, then you have to copy the vector. But if the caller passes an rvalue reference, then you can steal the vector by moving it into the `m_values`.

But really, the extra overload is just duplicated code for no real purpose. You're going to keep the object either way, so just make the caller provide it in a form you can keep: Pass it by value.

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

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

If the caller tries to pass an lvalue reference, then the compiler will use the copy constructor to create the inbound `value` parameter, which you then move into `m_values`.

If the caller tries to pass an rvalue reference, then the compiler will move it into the inbound `value` parameter, which you then move onward into `m_values`.

You get the same result without needing any overloads. Lvalues are copied and rvalues are moved.[1]

Related reading: Rvalue references in constructor: when less is more.

[1] Now, there is a bit of extra cost associated with this simplification: The copying or moving into the parameter is done at the call site, and the formal parameter now needs to be destructed. But you avoid having to write two versions of every function, and I think that simplification is worth it. (Besides, the compiler can often optimize the move.)

Raymond Chen

Follow