07 APRIL 2016

C++ std::move and std::forward

C++ std::move does not move and std::forward does not forward. This article dives deep into a long list of rules on lvalues, rvalues, references, overloads and templates to be able to explain a few deceivingly simple lines of code using std::move and std::forward.

Motivation

N4543 suggests a solution for when the content of a `std::function` is not copyable. But if first starts with the code below (and I'm going to ignore the rest of N4543 here). It has a `commands` variable that maps strings to functions, and an utility function to insert a new command.

```
1  std::map<std::string, std::function<void()>> commands;
2
3  template<typename ftor>
4  void install_command(std::string name, ftor && handler)
5  {
6    commands.insert({
7      std::move(name),
8      std::forward<ftor>(handler)
9    });
10 }
```

The code above is easy to read, but has subtleties. The goal of the article is to provide enough background information to be able to understand in detail each line, in particular why does it use `std::move` in one place (at line 7) and `std::forward` in another (at line 8).

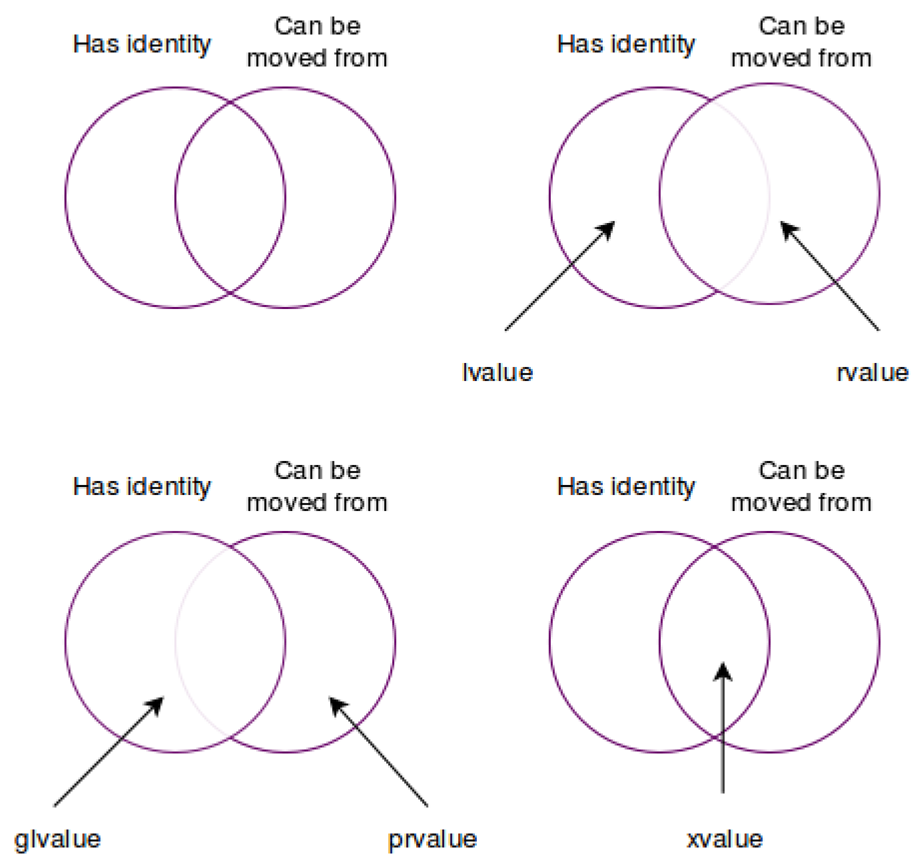
There is a lot of background information required. We'll start with the almost mathematical theory, but then dive quickly into C++ specifics and historical artefacts.

Value categories (lvalue, prvalue and xvalue)

A C++ expression has, in addition to a type, a value category. Traditionally the main value categories were `lvalue` and `rvalue` with a rough meaning that it if could stand on the left side of an assignment it's an `lvalue`, otherwise it's an `rvalue`.

With the advent of C++ 11, additional value categories have been identified and organized in a systematic way based on the observation that **there are two important properties of an expresion: has identity (i.e. 'can we get its address') and can be moved from**.

The naming of the main value categories is illustrated using Venn diagrams below.



- If it has identity, but cannot be moved it's an `lvalue`; otherwise it's an `rvalue`. A typical `lvalue` is a variable name `a`.
- If it can be moved, but has no identity is a `prvalue` (pure right value); otherwise it's a `glvalue` (generalized left value). A typical `prvalue` is a temporary resulting from a function call/operator (with a non-reference return type) like `s.substr(1, 2)` or `a + b` or integral constant like `42`.
- If it has an identity and can be moved it's an `xvalue` (because that was considered strange, and `x` is a good prefix for weird things). A typical `xvalue` is `std::move(a)`.

The above categories are the main ones. [There are additional ones](#) (e.g. `void` has a category with no identity and that can't be moved from), but I'm going to skip over them in this article.

References as function parameters (are lvalues)

References as function parameters are relevant here because they allow us to bind to arguments depending on their value category.

```
// i is a function parameter
void fn(int i) { }

int main() {
  int j = 42;
  // j is a function argument
  fn(j);
}
```

There are two types of reference declarations in C++. The pre-C++ 11 is called now `lvalue reference` (and uses one `&`), and the new C++ 11 called `rvalue reference` (that looks like `&&`).

If a function has `lvalue reference` parameter, then it can be called with an `lvalue` argument, but not an `rvalue` argument.

```
// parameter is lvalue reference
void fn(X &) { std::cout<< "X &\n"; }

int main()
{
  X a;
  fn(a); // works, argument is an lvalue

  fn(X()); // compiler error, argument is an rvalue
}
```

Similarly if a function has a `rvalue reference` parameter, then it can be called with an `rvalue` argument, but not an `lvalue` argument.

```
// parameter is rvalue reference
void fn(X &&) { std::cout<< "X &&\n"; }

int main()
{
  X a;
  fn(a); // compiler error, argument is an lvalue

  fn(X()); // works, argument is an rvalue
}
```

But **when used inside the function body, a parameter, whether `lvalue reference` or `rvalue reference`, is an `lvalue` itself: it has a name like any other variable**.

```
// parameter is rvalue reference
void fn(X && x)
{
  // but here expression x has an lvalue value category
  // can use std::move to convert it to an xvalue
}
```

From here, things get even more complicated when one notices that `const` matters. In the example below, `fn` can be called with both an `lvalue` and an `rvalue` argument. This is pre-C++ 11 behaviour that is unchanged.

```
// parameter is const rvalue reference
void fn(const X &) { std::cout<< "const X &\n"; }

int main()
{
  X a;
  fn(a); // works, argument is an lvalue

  fn(X()); // also works, argument is an rvalue
}
```

References and function overloads

We could provide overloads for `fn`, and we end up with three main overload options. If for an expression the preferred overload is not available, there is a fallback mechanism until all options are exhausted, and then we get a compiler error.

```
struct X {};

// overloads
void fn(X &) { std::cout<< "X &\n"; }
void fn(const X &) { std::cout<< "const X &\n"; }
void fn(X &&) { std::cout<< "X &&\n"; }

int main()
{
  X a;
  fn(a);
  // lvalue selects fn(X &)
  // fallbacks on fn(const X &)

  const X b;
  fn(b);
  // const lvalue requires fn(const X &)

  fn(X());
  // rvalue selects fn(X &&)
  // and then on fn(const X &)
}
```

In addition to the three overloads above there is of course the option of the overload with a `const X &&` argument, but I'm going to skip over it in this article.

A typical usage of this rules is for the typical copy/move constructor/assignment quadruple. For a user defined class `X` we expect two overloads requiring:

- `const X &` for copy constructor or assignment
- `X &&` for the move constructor or assignment

Template argument deduction and reference collapsing rules

If a templated function declares an argument as an `rvalue reference` to one of its template parameters, special template argument deduction rules kick in. Despite the syntactic similarities with the `rvalue reference` rules above, the rules for this case were specifically designed to support argument forwarding and are called forwarding references.

```
template<typename T>
void foo(T &&); // forwarding reference here
// T is a template parameter for foo

template<typename T>
void bar(std::vector<T> &&); // but not here
// std::vector<T> is not a template parameter,
// only T is a template parameter for bar
```

The rules allow the function `foo` above to be called with either an `lvalue` or an `rvalue`:

- When called with an `lvalue` of type `X`, then `T` resolves to `X &`

- When called with and `rvalue` of type `X`, then `T` resolves to `X`

When applying these rules we end up with an argument being `X & &&`. So there are even more rules to collapse the outcome:

- `X & &` collapses to `X &`
- `X & &&` collapses to `X &`
- `X && &` collapses to `X &`
- `X && &&` collapses to `X &&`

Combining the two rules we can have:

```
template<typename T>
void fn(T &&) { std::cout<< "template\n"; }

int main()
{
  X a;
  fn(a);
  // argument expression is lvalue of type X
  // resolves to T being X &
  // X & && collapses to X &

  fn(X());
  // argument expression is rvalue of type X
  // resolves to T being X
  // X && stays X &&
}
```

static_cast<X &&>

Once we have an expression of a value category, we can convert it to an expression of a different value category. If we have a `rvalue` we can assign it to a variable, or take a reference, hence becoming a `lvalue`. If we have a `lvalue` we can return it from a function, so we get a `rvalue`.

But one important rule is that: **one can covert from a `lvalue` to a `rvalue` (to an `xvalue` more precisely) by using static_cast<X &&> without creating temporaries**. And this is the last piece of the puzzle to understand `std::move` and `std::forward`.

std::move

The idiomatic use of `std::move` is to ensure the argument passed to a function is an `rvalue` so that you can move from it (choose move semantics). By function I mean an actual function or a constructor or an operator (e.g. assignment operator).

Here is an example where `std::move` is used twice in an idiomatic way:

```
struct X
{
  X(){}

  X(const X & other) : s_{ other.s_ } {}

  X(X && other) : s_{ std::move(other.s_) } {}
```

```
  // other is an lvalue, and other.s_ is an lvalue too
  // use std::move to force using the move constructor for s_
  // don't use other.s_ after std::move (other than to destruct)

  std::string s_;
};

int main()
{
  X a;

  X b = std::move(a);
  // a is an lvalue
  // use std::move to convert to a rvalue,
  // xvalue to be precise,
  // so that the move constructor for X is used
  // don't use a after std::move (other than to destruct)
}
```

Here is a [possible implementations](#) for `std::move`.

```
template<typename T> struct remove_reference { typedef T type; };
template<typename T> struct remove_reference<T&> { typedef T type; };
template<typename T> struct remove_reference<T&&> { typedef T type; };

template<typename T>
constexpr typename remove_reference<T>::type && move(T && arg) noexcept
{
  return static_cast<typename remove_reference<T>::type &&>(arg);
}
```

First of all `std::move` is a template with a `forwarding reference` argument which means that it can be called with either a `lvalue` or an `rvalue`, and the reference collapsing rules apply.

Because the type `T` is deduced, we did not have to specify when **using** `std::move`.

Then all it does is a `static_cast`.

The `remove_reference` template specializations are used to get the underlying type for `T` without any references, and that type is decorated with `&&` for the `static_cast` and return type.

In conclusion `std::move` does not move, all it does is to return a `rvalue` so that the function that actually moves, eventually receiving a `rvalue reference`, is selected by the compiler.

std::forward

The idiomatic use of `std::forward` is inside a templated function with an argument declared as a `forwarding reference`, where the argument is now `lvalue`, used to retrieve the original value category, that it was called with, and pass it on further down the call chain (perfect forwarding).

Here is an example where `std::forward` is used twice in an idiomatic way:

```
struct Y
{
  Y(){}
  Y(const Y &){ std::cout << "Copy constructor\n"; }
  Y(Y &&){ std::cout << "Move constructor\n"; }
};

struct X
{
  template<typename A, typename B>
  X(A && a, B && b) :
    // retrieve the original value category from constructor call
    // and pass on to member variables
    a_{ std::forward<A>(a) },
    b_{ std::forward<B>(b) }
  {
  }

  Y a_;
  Y b_;
};

template<typename A, typename B>
X factory(A && a, B && b)
{
  // retrieve the original value category from the factory call
  // and pass on to X constructor
  return X(std::forward<A>(a), std::forward<B>(b));
}

int main()
{
  Y y;
  X two = factory(y, Y());
  // the first argument is a lvalue, eventually a_ will have the
  // copy constructor called
  // the second argument is an rvalue, eventually b_ will have the
  // move constructor called
}

// prints:
// Copy constructor
// Move constructor
```

Here is a possible implementations for `std::forward`.

```
template<typename T> struct is_lvalue_reference { static constexpr bool value = false; };
template<typename T> struct is_lvalue_reference<T&> { static constexpr bool value = true; };

template<typename T>
constexpr T&& forward(typename remove_reference<T>::type & arg) noexcept
{
  return static_cast<T&&>(arg);
}

template<typename T>
constexpr T&& forward(typename remove_reference<T>::type && arg) noexcept
```

```
{
  static_assert(!is_lvalue_reference<T>::value, "invalid rvalue to lvalue conversion");
  return static_cast<T&&>(arg);
}
```

First of all `std::forward` is more complex than `std::move`. This version is the [result of several iterations](#).

The type `T` is not deduced, therefore we had to specify it when **using** `std::forward`.

Then all it does is a `static_cast`.

The `static_assert` is there to stop at compile time attempts to convert from an `rvalue` to an `lvalue` (that would have the dangling reference problem: a reference pointing to a temporary long gone). This is explained in more details in [N2835](#), but the gist is:

```
forward<const Y&>(Y()); // does not compile
// static assert in forward triggers compilation failure for line above
// with "invalid rvalue to lvalue conversion"
```

Some non-obvious properties of `std::forward` are that the return value can be more cv-qualified (i.e. can add a `const`). Also it allows for the case where the argument and return are different e.g. to forward expressions from derived type to it's base type (even some scenarios where the base is derived from as `private`).

Conclusion

Going back to the code we started with:

```
std::map<std::string, std::function<void()>> commands;

template<typename ftor>
void install_command(std::string name, ftor && handler)
{
  commands.insert({
    std::move(name),
    std::forward<ftor>(handler)
  });
}
```

The first parameter, `name`, for the function `install_command` is passed [by value](#). That is really a temporary, but has a name, hence it's an `lvalue` expression inside `install_command`. The second parameter `handler` is a `forwarding reference`. Because it has a name, it's an `lvalue` expression as well inside `install_command`.

The `std::map` has an `insert` overload that accepts an templated `rvalue reference` for the key/value pair to insert. For the key we can provide an `rvalue` using `std::move` because really we don't need `name` any more. If we did not use `std::move` we would do a silly copy. For the value we provide whatever we the `install_command` was called with for the `handler`. We use `std::forward` to retrieve the original value category. If for the `handler` we provided an `rvalue` then `insert` will move from it. If for the `handler` we provided an `lvalue` then `insert` will copy it.

There are a lot of rules that come into play for the initial deceivingly simple code. They are the result of maintaining backward compatibility and plumbing move semantics and perfect forwarding support on top of that, while making it so that most common scenarios are easy to write and read.

References

- Stroustrup on C++ 11 value category classification: http://www.stroustrup.com/terminology.pdf
- Cppreference with details on value category: http://en.cppreference.com/w/cpp/language/value_category
- Thomas Becker on rvalue references: http://thbecker.net/articles/rvalue_references/section_08.html
- MSDN on rvalue references: https://msdn.microsoft.com/en-us/library/dd293668.aspx
- Final C++ 11 versions of std::forward and std::move: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3143.html
- Use cases for std::forward: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2951.html
- Explaining the static_assert in std::forward: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2835.html
- On reference binding rules: http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2008/n2812.html
- Howard Hinnant on Stack Overflow: http://stackoverflow.com/a/9672202/5495780

‹ **Special class operations. Comparing**

Dependency Injection Using Templates and Concepts ›