Программирование на языке C++
Лекция 8

Как работают rvalue-ссылки

Александр Смаль

Преобразование ссылок в шаблонах

“Склейка” ссылок:

- `T& &` $\rightarrow$ `T&`
- `T& &&` $\rightarrow$ `T&`
- `T&& &` $\rightarrow$ `T&`
- `T&& &&` $\rightarrow$ `T&&`

Универсальная ссылка

```
template<typename T>
void foo(T && t) {}
```

- Если вызвать `foo` от lvalue типа A, то `T` = A&.
- Если вызвать `foo` от rvalue типа A, то `T` = A.

Как работает std::move?

Определение std::move:

```
template<class T>
typename remove_reference<T>::type&&
   move(T&& a)
{
   typedef typename remove_reference<T>::type&& RvalRef;
   return static_cast<RvalRef>(a);
}
```

Замечание

std::move не выполняет никаких действий времени выполнения.

std::move для lvalue

Вызываем std::move для lvalue объекта.

```
X x;
x = std::move(x);
```

Тип T выводится как X&.

```
typename remove_reference<X&>::type&&
    move(X& && a)
{
    typedef typename remove_reference<X&>::type&& RvalRef;
    return static_cast<RvalRef>(a);
}
```

После склейки ссылок получаем:

```
X&& move(X& a)
{
    return static_cast<X&&>(a);
}
```

std::move для rvalue

Вызываем std::move для временного объекта.

```
X x = std::move(X());
```

Тип T выводится как X.

```
typename remove_reference<X>::type&&
   move(X&& a)
{
   typedef typename remove_reference<X>::type&& RvalRef;
   return static_cast<RvalRef>(a);
}
```

После склейки ссылок получаем:

```
X&& move(X&& a)
{
   return static_cast<X&&>(a);
}
```

Perfect forwarding

```
// для lvalue
template<typename T, typename Arg>
unique_ptr<T> make_unique(Arg & arg) {
    return unique_ptr<T>(new T(arg));
}

// для rvalue
template<typename T, typename Arg>
unique_ptr<T> make_unique(Arg && arg) {
    return unique_ptr<T>(new T(std::move(arg)));
}
```

`std::forward` позволяет записать это одной функцией.

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

Как работает `std::forward`?

Определение `std::forward`:

```
template<class S>
S&& forward(typename remove_reference<S>::type& a)
{
    return static_cast<S&&>(a);
}
```

Замечание

`std::forward` не выполняет никаких действий
времени выполнения.

std::forward для lvalue

```
X x;
auto p = make_unique<A>(x);      // Arg = X&
```

```
unique_ptr<A> make_unique(X& && arg) {
  return unique_ptr<A>(new A(std::forward<X&>(arg)));
}

X& && forward(remove_reference<X&>::type& a) {
  return static_cast<X& &&>(a);
}
```

std::forward для lvalue

```
X x;
auto p = make_unique<A>(x);      // Arg = X&
```

```
unique_ptr<A> make_unique(X& && arg) {
  return unique_ptr<A>(new A(std::forward<X&>(arg)));
}

X& && forward(remove_reference<X&>::type& a) {
  return static_cast<X& &&>(a);
}
```

После склейки ссылок:

```
unique_ptr<A> make_unique(X& arg) {
  return unique_ptr<A>(new A(std::forward<X&>(arg)));
}

X& forward(X& a) {
  return static_cast<X&>(a);
}
```

std::forward для rvalue

```
auto p = make_unique<A>(X());    // Arg = X
```

```
unique_ptr<A> make_unique(X&& arg) {
  return unique_ptr<A>(new A(std::forward<X>(arg)));
}

X&& forward(remove_reference<X>::type& a) {
  return static_cast<X&&>(a);
}
```

std::forward для rvalue

```
auto p = make_unique<A>(X());    // Arg = X
```

```
unique_ptr<A> make_unique(X&& arg) {
  return unique_ptr<A>(new A(std::forward<X>(arg)));
}

X&& forward(remove_reference<X>::type& a) {
  return static_cast<X&&>(a);
}
```

После склейки ссылок:

```
unique_ptr<A> make_unique(X&& arg) {
  return unique_ptr<A>(new A(std::forward<X>(arg)));
}

X&& forward(X& a) {
  return static_cast<X&&>(a);
}
```

Variadic templates + perfect forwarding

Можно применить `std::forward` для списка параметров.

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

Теперь `make_unique` работает для произвольного числа аргументов.

```
auto p = make_unique<Array<string>>(10, string("Hello"));
```