Лекция 2. Семантика перемещения

ИУ8

September 15, 2016

Содержание

На текущей лекции рассмотрим:

- rvalue references
- move semantic
- RVO
- universal references
- perfect forwarding

Мотивация

Семантика перемещения позволяет компиляторам заменять дорогостоящие операции копирования менее дорогими перемещениями. Перемещающие конструкторы и перемещающие операторы присваивания предоставляют контроль над семантикой перемещения.

Семантика перемещения позволяет также создавать типы, которые могут только перемещаться, такие как std::unique_ptr, std::future или std::thread.

Прямая передача делает возможным написание шаблонов функций, которые принимают произвольные аргументы и передают их другим функциям так, что целевые функции получают **в точности** те же аргументы, что и переданные исходным функциям.

Rvalue-ссылки делают возможными как семантику перемещения, так и прямую передачу.

Мотивация

Пример

```
1 // old style swap
2 template<class T> swap(T& a, T& b)
3 {
4   T tmp(a); // now we have two copies of a
5   a = b;  // now we have two copies of b
6   b = tmp; // now we have two copies of tmp (aka a)
7 }
```

Мнение профессионала

Perhaps the most significant new feature in C++11 is rvalue references (*Scott Meyers*)

Семантика перемещения, пожалуй, самая главная возможность C++11

Интуитивное определение

lvalue это выражение, которое может появлятьсся **слева** или **справа** от знака присваивания

rvalue это выражение, которое может появлятьсся **ТОЛЬКО справа** от знака присваивания

Альтернативное определение

lvalue это выражение, которое ссылается на область памяти и к которому применима операция получения адреса выражения

rvalue это НЕ **lvalue** это выражение

Пример

```
int a = 42;
int b = 43;

// a and b are both l-values:
a = b; // ok
b = a; // ok
a = a * b; // ok
```

```
// a * b is an rvalue:
int c = a * b; // ok
a * b = 42; // error

&a++; // error
&++a; // ok
```

```
// lvalues:

int i = 42;
i = 43; // ok
int* p = &i; // ok
int& foo();
foo() = 42; // ok
int* p1 = &foo(); // ok
```

```
// rvalues:
int foobar();
int j = 0;
j = foobar(); // ok

//can't take address of rvalue
int* p2 = &foobar(); //error
j = 42; // ok
```

Определение

Пусть X любой конкретный тип, тогда X&& называется rvalue ссылкой на тип X.

Семантика перемещений

Пусть X - класс, котрый владеет указателем (**pResource**) на некоторый *ресурс*. Здесь под *ресурсом* понимаем всё, что требует значительных усилий при создании, скопировании и удалении.

Рассмотрим пример

```
1 X foo();
2 X x;
3 // use x in various ways
4 x = foo();
```

Что происходит в последней строке

Копируется ресурс из временного объекта, возвращенного из *foo()*:

- вызывается деструктор ресурса в объекте **x**
- ресурс объекта **x** заменяется на копию ресурса из временного объекта
- вызывается деструктор временного объекта, тем самым освобождается его ресурс

Семантика перемещений

Оператор присваивания класса X

```
1 X& X::operator=(X const & rhs)
2 {
3   // ...
4   // Make a clone of what rhs.pResource refers to.
5   // Destruct the resource that pResource refers to.
6   // Attach the clone to pResource.
7   // ...
8 }
```

Очевидно, что в нашем примере эфективней использовать другую схему "оператора присваивания". А именно в случае, когда аргумент оператора является rvalue, нам необходимо просто поменять местами указатели на ресурсы.

С использованием сематики перемещения

```
1 X& X::operator=(X && rhs)
2 {
3   // ...
4   std::swap(this->pResource, rhs.pResource);
5   // ...
6 }
```

Rvalue ссылка

Перегрузка функции

```
1 void foo(X& x); // lvalue reference overload
2 void foo(X&& x); // rvalue reference overload
3
4 X x;
5 X foobar();
6
7 foo(x); // argument is lvalue: calls foo(X&)
8 foo(foobar()); // argument is rvalue: calls foo(X&&)
```

Отметим случаи

```
1 void foo(X& x);
2 //void foo(X&& x);
3 foo(x); // ok
4 foo(foobar()); // ok
```

```
1 void foo(X const& x);
2 //void foo(X&& x);
3 foo(x); // ok
4 foo(foobar()); // ok
```

```
1 //void foo(X& x);
2 void foo(X&& x);
3 foo(x); // error
4 foo(foobar()); // ok
```

New style swap

C++98 and std::swap

```
// old style swap
template<class T>
void swap(T& a, T& b)
{
  T tmp(a);
  a = b;
  b = tmp;
}
```

```
X a, b;
swap(a, b);
// a, b is lvalue!
```

C++11 and std::move

```
// new style swap
template<class T>
void swap(T& a, T& b)
{
  T tmp(std::move(a));
  a = std::move(b);
  b = std::move(tmp);
}
```

RVO

Не переусердствуй!

```
1 // good sample
2 Widget makeWidget() {
3   Widget w;
4   // ...
5   return w;
6 }
```

```
1 // bad sample
2 Widget makeWidget() {
3   Widget w;
4   // ...
5   return std::move(w);
6 }
```

Никогда не применяйте std::move и std::forward к локальным объектам, которые могут быть объектом оптимизации возвращаемого значения

std::move

Исходный код C++14

```
1 template<typename T>
2 decltype(auto) move(T&& arg) {
3   using ReturnType = remove_reference_t<T>&&;
4   return static_cast<ReturnType>(arg);
5 }
```

- std::move ничего не перемещает!
- std::move во время выполнения ничего не делает!
- std::move не генерирует выполнающийся код!
- std::move выполняет безусловное примедение агрумента к rvalue

- Не объявляйте объекты константными, если хотите иметь возможность их перемещать
- std::move не гарантирует, что приведенный объект будет иметь право быть перемещенным

Вопрос к аудитории

```
void foo(X&& x)
{
  X anotherX = x;
  // ...
}
```

Какой конструктор вызовится: копирования или перемещения?

Критерий rvalue

if it has a name, then it is an lvalue. Otherwise, it is an rvalue.

Следовательно будет вызван конструктор **копирования**, а не перемещения.

Какой вывод надо сделать?

```
Base(Base const & rhs);
Base(Base&& rhs);

Derived(Derived&& rhs)
  : Base(rhs)
  // wrong: rhs is an lvalue
{
  // Derived-specific stuff
}
```

```
Base(Base const & rhs);
Base(Base&& rhs);

Derived(Derived&& rhs)
  : Base(std::move(rhs))
  // good, calls Base(Base&& rhs)
{
  // Derived-specific stuff
}
```

Bad case

```
1 Widget w;
2 // use w - OK
3 w.method();
4 Widget w2{std::move(w);}
5 // use w2 - OK
6 w2.method();
7 // use w - Bad idea
8 w.method();
```

normal case

```
1 Widget w;
2 // use w - OK
3
4 Widget w2{std::move(w);} // after this never use w!!!!!
5 // use w2 - OK
```

Универсальная ссылка

Определение rvalue ссылки

Пусть X любой конкретный тип, тогда X&& называется rvalue ссылкой на тип X.

```
1  class Widget {/*...*/};
2  void f(Widget &&); // rvalue reference
3  Widget&& var1 = Widget(); // rvalue reference
4
5  auto&& var2 = var1; // isn't rvalue reference
6
7  template<typename T>
8  void f(std::vector<T>&& param); // rvalue reference
9
10 template<typename T>
11 void f(T&& param); // isn't rvalue reference
```

Универсальная ссылка

Определение rvalue ссылки

Пусть X любой **конкретный** тип, тогда X&& называется rvalue ссылкой на тип X.

```
1  class Widget {/*...*/};
2  void f(Widget &&); // rvalue reference
3  Widget&& var1 = Widget(); // rvalue reference
4
5  auto&& var2 = var1; // universal reference
6
7  template<typename T>
8  void f(std::vector<T>&& param); // rvalue reference
9
10 template<typename T>
11 void f(T&& param); // universal reference
```

Универсальная ссылка

Определение универсальной ссылки для студентов

Универсальной ссылкой называют сущность, которая может быть связана с rvalue объектами, с lvalue объектами. Также они могут быть связаны с константными и неконстантными объектами, с объектами объявленными volatile и с объектами необъявленными volatile.

Универсальная ссылка

Случаи возникновения универсальной ссылки

Универсальные ссылки возникают в случае, при котором происходит вывод типа! Вывод типа необходим для того, чтобы ссылка была универсальной, но не достаточен.

```
1 auto&& var2 = var1; // universal reference
2
3 template<typename U>
4 void f(U&& param); // universal reference
```

Ограничение на возникновении универсальной ссылки

Объявления универсальной ссылки должен в точности иметь вид "T&&".

```
1 template<typename T>
2 void f(std::vector<T>&& param); // rvalue reference
```

Универсальная ссылка

Вопрос к аудитории

```
template<class T>
class container {
public:
  // What type of reference is T&& ?
  void push_back(T&& x);
};
```

Ответ: rvalue reference!

Вопрос к аудитории

```
template<class T>
class container {
public:
  // What type of reference is Args&& ?
  template<class ... Args>
  void emplace_back(Args&&... args);
};
```

Ответ: universal reference!

Универсальная ссылка

Как универсальная ссылка становится rvalue/lvalue ссылкой?

Любая ссылка должна быть проинициализирована. Инициализатор универсальной ссылки определяет, какую ссылку она представляет. Если инициализатор представляет собой rvalue, универсальная ссылка соответствует rvalue ссылке. Если инициализатор представляет собой lvalue, универсальная ссылка соответствует lvalue ссылке.
Для универсальных ссылок, которые являются параметрами фунций, инициализатор предоставляется в месте вызова.

```
1 template<typename T>
2 void f(T&& param); // param is universal reference
3
4 Widget w;
5 f(w); // param is Widger& (lvalue reference)
6
7 f(std::move(w)); // param is Widget&& (rvalue reference)
```

Универсальная ссылка

- Если параметр шаблона функции имеет тип **T&&** для выводимого типа **T** или если объект объявлен с использованием **auto&&**, то параметр или объект является универсальной ссылкой
- если вид объявления типа не является в точности **type&&** или если вывод типа не имеет места, то **type&&** означает rvalue-ссылку
- увеверсальные ссылки соответсвуют rvalue-ссылкам, если они инициализируются значение rvalue. Если универсальные ссылки инициализируются значением lvalue, то они соответсвуют lvalue-ссылкам

Perfect forwading

Задача о прямой передачи

Требуется реализовать функцию, которая передаст принимаемые параметры в другую функцию, не создавая временные переменные, то есть выполнит **прямую передачу**.

Первый вариант "решения"

```
1 struct Type{
2   Type(/* some args */);
3 };
4
5 template<typename Arg1, typename Arg2>
6 Type factory(Arg1 a, Arg2 b) {
7   return Type(a, b);
8 }
```

Perfect forwading

Задача о прямой передачи

Требуется реализовать функцию, которая передаст принимаемые параметры в другую функцию, не создавая временные переменные, то есть выполнит **прямую передачу**.

OK, решим проблему из первого "решения" - добавим ссылки

```
1 struct Type{
2   Type(/* some args */);
3 };
4
5 template<typename Arg1, typename Arg2>
6 Type factory(Arg1& a, Arg2& b) {
7   return Type(a, b);
8 }
```

```
1 factory(42, 3.14f); // error: invalid initialization of non-const
      reference from an rvalue
2 factory(foo(), foo()); // error: invalid initialization of non-const
      reference from an rvalue
```

Будем пробовать константные ссылки? Нет, и они не помогут

Perfect forwading

Bruteforce

```
1  template <typename T1, typename T2>
2  Type factory(T1& e1, T2& e2) {
3    return Type(e1, e2);
4  }
5
6  template <typename T1, typename T2>
7  Type factory(const T1& e1, T2& e2) {
8    return Type(e1, e2);
9  }
10
11 template <typename T1, typename T2>
12 Type factory(T1& e1, const T2& e2) {
13   return Type(e1, e2);
14 }
15
16 template <typename T1, typename T2>
17 Type factory(const T1& e1, const T2& e2) {
18   return Type(e1, e2);
19 }
```

А теперь реализуйте для класса **Type**, который принимает 6 аргуметов....

Perfect forwading

Свертывание ссылок

```
template <typename T>
void foo(T t) {
  T& k = t;
}
// ...
int lvalue = 4;
foo<int&>(lvalue);
```

Правило свертывания ссылок

Если любая из ссылок является lvalue-ссылкой, результат представляет собой lvalue-ссылку. В противном случае (т.е. когда обе ссылки являются rvalue-ссылками) результат представляет собой rvalue-ссылку.

```
Type& & becomes Type&
Type& && becomes Type&
Type&& & becomes Type&
Type&& && becomes Type&&
```

Perfect forwading

При чем тут универсальные ссылки

```
1 template<typename T>
2 void foo(T&& arg);
```

- Когда foo вызывается с lvalue типа A, то T инстанцируется как A&, а по правилу свертывания ссылок, тип аргумента становится A&
- Когда foo вызывается с rvalue значением, то T инстанцируется A, а по правилу свертывания ссылок, тип аргумента становится A&&

Решение задачи о прямой передачи

```
1 template<typename Arg1,
2      typename Arg2>
3 Type factory(Arg1&& a,
4        Arg2&& b) {
5   return Type(
6     std::forward<Arg1>(a),
7     std::forward<Arg2>(b)
8   );
9 }
```

```
1 // use variadic templates
2
3 template<typename... Args>
4 Type factory(Args&&... args)
5 {
6   return Type(
7     std::forward<Args>(args)...
8   );
9 }
```

Perfect forwading

std::forward

```
1 template<class T>
2 T&& forward(remove_reference_t<T> & a)
3 {
4   return static_cast<T&&>(a);
5 }
```

- std::forward ничего не передает!
- std::forward во время выполнения ничего не делает!
- std::forward не генерирует выполнающийся код!
- std::forward выполняет приведение только при соблюдении определенных условий

Perfect forwading

Как работает Perfect forwading: lvalue

```
X x;
factory(x);
```

```
Type factory(X& && arg) {
  return Type(std::forward<X&>(arg));
}

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

Брюки превращаются...

```
Type factory(X& arg) {
  return Type(std::forward<X&>(arg));
}

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

Perfect forwading

Как работает Perfect forwading: rvalue

```
1 X foo();
2 factory(foo());
```

```
1 Type factory(X && arg) {
2   return Type(std::forward<X>(arg));
3 }
4
5 X&& forward(remove_reference_t<X>& a) {
6   return static_cast<X&&>(a);
7 }
```

Таким образом rvalue аргументы передаются, как rvalue ссылки, а lvalue аргументы передаются, как lvalue ссылки.

The end

Самостоятельное изучение

- Методы emplace_back и emplace для стандартных контейнеров
- Когда сематика перемещения не работает
- Случаи некорректной работы прямой передачи
- Почему стоит избегать перегрузки универсальных ссылок
- Паттерны проектирования

Список литературы

- С. Мэйерс - Эффективный и современный C++
- https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers
- http://thbecker.net/articles/rvalue_references/section_01.html
- https://accu.org/index.php/journals/227
- https://www.ibm.com/developerworks/community/blogs/5894415f-be62-4bc0-81c5-3956e82276f3/entry/RVO_V_S_std_move?lang=en
- http://alenacpp.blogspot.ru/2008/02/rvo-nrvo.html