rvalue ссылки и изменения, которые они привносят в C++

2. августа 2011 разработка

Прежде чем говорить о новшествах, необходимо раскрыть тему *свойств выражений*. Итак, начнем, - lvalue и rvalue существуют достаточно давно, но не каждый C++ программист подозревает об их существовании и еще меньшая часть из них сможет с ходу определить какое выражение относиться к rvalue, а какое к lvalue. Необходимо знать, что lvalue и rvalue это свойство **выражения,** некоторые ошибочно полагают, что переменные или объекты имеют свойства rvalue\lvalue, но это предположение ложно т.к. **только выражения** обладают подобными свойствами.

Для понимания сути lvalue\rvalue можно заглянуть в историю появления их непрезентабельных имен: свои имена они получили благодаря фразам: right value(rvalue), т.е. выражение находящиеся справа и left value(lvalue), т.е. выражение находящееся слева. Это очень грубое объяснение сути rvalue\lvalue, но зато, их имена прекрасно отражают суть изначальной задумки комитета стандартизации. Со времени появления lvalue\rvalue много воды утекло, и сейчас их уже нельзя разделить на категории находящихся справа и слева, ведь для того, чтобы определить их местоположения, надо знать, относительно чего определять это самое местоположение. И тут нет никакого общего правила, ведь, например, слева от оператора '.'(точка) может быть как lvalue, rvalue выражение, а следовательно нельзя говорить, что свойство выражения определяется его пространственным расположением.

Ну да ладно, хватит уроков истории пора перейти к практике. Простейшим способом определения является попытка получения адреса выражения; если Вы можете получить его адрес и в дальнейшем его использовать, тогда перед вами lvalue. Если же адрес выражения не может быть получен - перед вами rvalue. Кто-то может поспорить, что адрес все-таки можно получить, и некоторые компиляторы, возможно, дают это сделать. Тем не менее, это является нарушением стандарта C++ а именно: пункта 5.3.1/3. Да и если рассуждать логически: этот адрес не будет иметь никакого смысла, так как это адрес памяти, которую вы не контролируете и она может быть легко перезаписана в течение работы программы. Адрес же lvalue, это адрес постоянного хранилища, которое остается под контролем программиста на протяжении всей области жизни объекта.

Пример:

```
1  int a = 0, b = 0;
2  /*
3      Можем ли мы получить адрес выражения (a + b) и использовать е
4      в дальнейшем? Нет. Т.к. результатом сложения будет временный
5      объект, доступ к которому не может быть получен за пределами
       выполнения оканчивающейся ';' (точкой с запятой).
6  */
7  (a + b);
8  /*
9      Мы можем получить адрес результата этого выражения использова
10     его в дальнейшем т.к. адресом этого выражения будет служить
       адрес переменной a.
11 */
12 a += b;
```

```
13
14
```

Еще один, более развернутый, пример:

```
1  int foo();
2  int& bar();
3
4  int main()
5  {
6      int i = 0;
7      &++i;//Ok, lvalue
8      &i++;//error C2102: '&' requires l-value(VC++ 2010)
9      &foo();//error C2102: '&' requires l-value(VC++ 2010)
10     &bar();//Ok, lvalue
11 }
12
13 int foo()
14 {
15     return 0;
16 }
17
18 int& bar()
19 {
20     static int value = 0;
21     return value;
22 }
```

Хотя приведенное выше методика распознавания rvalue не универсальна, я считаю, что она может стать хорошим подспорьем для людей не искушенных в тонкостях C++ и позволит выработать рефлекс на rvalue. Однако, хотелось бы предостеречь читателя от искушения считать временный объект и rvalue синонимами, еще раз напомню rvalue\lvalue это*свойство выражения*, а не объекта. Например:

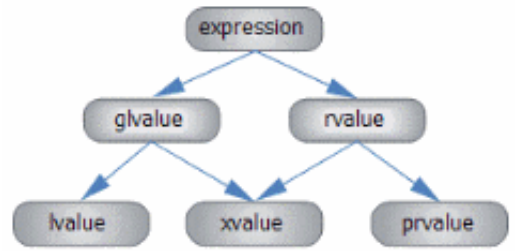
```
1  foo();
```

Результатом выполнения функции будет временный, безымянный объект, который будет уничтожен в конце выражения, т.е. это выражение является rvalue. Теперь немного модифицируем пример:

```
1  const int& lvalue = foo();
```

Объект возвращаемый foo() все еще является временным, но, согласно пункту 12.2/5 стандарта C++, время жизни временного объекта продлевается и становится таким же, как и время жизни ссылки, которая указывает на этот объект. Выражение же, в свою очередь, приобрело свойство lvalue. Таким образом, пример приведенный выше показывает, что нельзя ставить знак равенства между временным объектом и rvalue.

Пожалуй мы разобрались с lvalue\rvalue которые существуют в нынешнем стандарте C++03. Пора перейти к тому, что же предлагает по этому поводу новый стандарт. А предлагает он следующее разделение:



Новый стандарт вводит новые понятия, и расширяет идею rvalue\lvalue. Что же значат эти новые понятия? Давайте разберемся с каждым по очереди:

lvalue - здесь никаких изменений, lvalue свойство обозначает тоже самое, что обозначает в C++03.

xvalue(от английского "eXpiring"(находящийся на грани уничтожения)) - это свойство означает, что объект(результат выражения) находится в конце своего жизненного цикла, но еще не удален. **xvalue**появляется в тех выражениях, результат которых связан с *rvalueссылками*(о них мы поговорим позже)

prvalue(pure(англ. чистый) "rvalue") - в новом стандарте так обозвали rvalue из C++03.

rvalue - это свойство которое делится на **xvalue** и **prvalue**

glvalue - это свойство которое делится на **xvalue** и **lvalue**

Таким образом, в грядущем стандарте систему свойств выражений несколько усложнили. Хотя, если вы усвоили rvalue\lvalue свойства из C++03 то особых проблем с новыми свойствами у вас возникнуть не должно. Ведь lvalue и prvalue вам уже знакомы, а xvalue выходит на сцену лишь, когда в результате выражения фигурируют *rvalue ссылки.* Давайте поговорим об этих самых ссылках.

rvalue ссылки

Одним из самых значительных изменений в ядре языка по праву можно считать введение rvalue ссылок. Они получили свое имя по аналогии с давно знакомыми любому C++ программисту ссылками. Теперь эти, "старые" ссылки именуются не иначе как lvalue ссылки. Чем же отличаются rvalue ссылки от своих предшественниц lvalue ссылок? Во-первых, отличие в способе записи, если lvalue ссылки используют лексему "**&**"(амперсанд) для своей декларации, то rvalue использует двойной амперсанд "**&&**". Кто-то может воскликнуть, так это же "ссылка на ссылку!"; да, это выглядит именно так и многие были недовольны подобной нотацией, считая, что это может смутить конечных пользователей. Но нас так просто не смутишь, правда? Во-вторых, они отличаются по типам объектов, на которые они могут ссылаться:

```
1 Type& //может ссылаться на любое не константное lvalue.
2 const Type& //может ссылаться на любое выражение.
3 Type&& //может ссылаться на не константные xvalue и prvalue.
4 const Type&& //может ссылаться на любое выражение, кроме lvalue.
```

Для лучшего усвоения, давайте рассмотрим следующий пример:

```
int&& xvalue_func();
int& lvalue_func();
int prvalue_func();


int main()
{
    double d = 0.0;
    const int i = 0;
    //---Type&
    //#1:Ok, простое lvalue
    int& lvalue = lvalue_func();
    //#2:Error, lvalue ссылка не может быть привязана к prvalue
    int& wrong_lvalue1 = prvalue_func();
    //#3:Error, lvalue ссылка не может быть привязана к xvalue
    int& wrong_lvalue2 = xvalue_func();
    /*
        #4:Error, lvalue ссылка на не константу, не может быть
        привязана к константному выражению
    */
    int& non_const_lvalue = i;
    /*
        #5:Error, lvalue ссылка не может быть привязана к перемен
        чей интегральный тип не совпадает с типом ссылки
    */
    int& type_mismatch_lvalue = d;
    //---Type&&
    //#6:Error, rvalue ссылка не может быть привязана к lvalue
    int&& rvalue1 = lvalue_func();
    //#7:Ok, rvalue ссылка привязывается к xvalue
    int&& rvalue2 = xvalue_func();
    //#8:Ok, rvalue ссылка привязывается к prvalue
    int&& rvalue3 = prvalue_func();
    //#9:Ok, rvalue ссылка привязывается к prvalue
    int&& rvalue4 = 0;
    /*
        #10:Error, rvalue ссылка на не константу, не может быть
        привязана к  константному выражению
    */
    int&& non_const_rvalue = i;
    /*
        #11:Ok, rvalue ссылка может быть привязана к переменной,
        интегральный тип не совпадает с типом ссылки
    */
    int&& type_mismatch_rvalue = d;
    //---const Type&
    //#12:Ok, const Type& может быть привязано к любому выражени
    const int& const_lvalue1 = lvalue_func();
    //#13:Ok, const Type& может быть привязано к любому выражени
    const int& const_lvalue2 = prvalue_func();
    //#14:Ok, const Type& может быть привязано к любому выражени
    const int& const_lvalue3 = xvalue_func();
    //#15:Ok, const Type& может быть привязано к любому выражени
    const int& const_lvalue4 = i;
    //#16:Ok, const Type& может быть привязано к любому выражени
    const int& const_lvalue5 = d;
    //#17:Ok, const Type& может быть привязано к любому выражени
    const int& const_lvalue6 = 0;
    //---const Type&&
    //#18:Error, const Type&& не может быть привязано к lvalue
    const int&& const_rvalue1 = lvalue_func();
    //#19:Ok, const Type&& может быть привязано к prvalue
    const int&& const_rvalue2 = prvalue_func();
    //#20:Ok, const Type&& может быть привязано к xvalue
    const int&& const_rvalue3 = xvalue_func();
```

```
    //#21:Error, const Type&& может быть привязано к lvalue
    const int&& const_rvalue4 = i;
    /*
        #22:Ok, const Type&& может быть привязано к выражению, че
        интегральный тип не совпадает с типом ссылки
    */
    const int&& const_rvalue5 = d;
}

int&& xvalue_func()
{
    return 5;
}

int& lvalue_func()
{
    static int i = 0;
    return i;
}

int prvalue_func()
{
    return 5;
}
```

Некоторые пункты, я считаю, требуют пояснения. Например **#5**: в этом пункте происходит следующее, выражение *d* является lvalue, а переменная *d* имеет тип double. В то же время ссылка имеет тип int. Таким образом, чтобы убрать различие в типах необходимо сконвертировать *d* в int. Но после конвертации получается временный объект типа int, и в результате выражение из lvalue превращается в prvalue! А, как мы уже знаем, lvalue ссылка не может быть привязана к prvalue. Это справедливо не только для интегральных типов, все это справедливо и для классов в той же мере. Для легкости определения таких "узких" мест можно пользоваться простым вопросом: "Что будет результатом выражения?". Еще один пример к этому пункту:

```
/*
    "Hello", непосредственно, является lvalue, но в результате
    исполнения выражения будет создан временный объект
    std::string и выражение приобретет тип prvalue
*/
std::string& lvalue_ref = "Hello!";
```

Вышеизложенное так же объясняет пункты **#11** и **#22**

Правила перегрузки

Помимо разницы в типах выражений, к которым могут быть привязаны lvalue и rvalue ссылки есть еще и различие в том, как они участвуют в перегрузке методов.

Здесь все достаточно просто:

1. lvalue жестко пытаются привязаться к lvalue ссылкам

2. rvalue жестко пытаются привязаться к rvalue ссылкам

3. Модифицируемые lvalue или rvalue мягко пытаются привязаться к не константным ссылкам на lvalue и rvalue соответственно.

Поясняющий пример:

```
#include <iostream>

int&& xvalue_func();
int& lvalue_func();
int prvalue_func();
const int const_prvalue_func();
void foo(int& arg);
void foo(const int& arg);
void foo(int&& arg);
void foo(const int&& arg);

int main()
{

    const int i = 0;
    foo(xvalue_func());
    foo(lvalue_func());
    foo(prvalue_func());
    foo(const_prvalue_func());
    foo(i);
    foo(5);
    std::cin.get();
}

void foo(int& arg)
{
    std::cout << "void foo(int& arg)" << std::endl;
}
```

```
25
26  void foo(const int& arg)
27  {
28      std::cout << "void foo(const int& arg)" << std::endl;
29  }
30
31  void foo(int&& arg)
32  {
33      std::cout << "void foo(int&& arg)" << std::endl;
34  }
35
36  void foo(const int&& arg)
37  {
38      std::cout << "void foo(const int&& arg)" << std::endl;
39  }
40
41  int&& xvalue_func()
42  {
43      return 5;
44  }
45
46  int& lvalue_func()
47  {
48      static int i = 0;
49      return i;
50  }
51
52  int prvalue_func()
53  {
54      return 5;
55  }
56
57  const int const_prvalue_func()
58  {
59      return 5;
60  }
61
62
63
64
```

Вывод:

```
1  void foo(int&& arg)
2  void foo(int& arg)
3  void foo(int&& arg)
4  void foo(const int&& arg)
5  void foo(const int& arg)
6  void foo(int&& arg)
```

Правило "свертки"

Вместо слов, я сразу приведу пример, чтобы понять к каким выражениям применяема свертка:

```
typedef int& IntRef;
void foo(IntRef&);
```

Таким образом мы имеем, что тип передаваемый в функции foo является ссылкой на ссылку или int& &(не путать с rvalue &&!), чего не может быть в текущей версии C++. Именно для таких ситуаций было придумано "правило свертки", применимое к typedef типам, шаблонам и типам, полученным помощью decltype. Введем понятие *выведенный тип*,для упрощения дальнейших объяснений. *Выведенный тип* - Это тип полученный посредством оператора *decltype*, определения с помощью оператора *typedef* или являющийся параметром шаблона.

Модификатором выведенного типа будем называть модификатор ссылки(& или &&), который используется в объявлении типа, например:

```
typedef int& IntRef;
template <typename T>
```

Правило можно выразить следующим образом: Если *выведенный тип*содержит модификатор rvalue ссылки, тогда результирующий тип будет являться rvalue ссылкой, тогда и только тогда, когда применяемый модификатор ссылки есть rvalue модификатор(&&), в противном случае результирующим типом будет являться lvalue ссылка. Если выведенный тип содержит модификатор lvalue ссылки, тогда какой-бы не был применен модификатор ссылки к выведенному типу результирующий тип останется lvalue ссылкой.

Пример применения правила:

	&	&&
int&	int&	int&
int&&	int&	int&&

Семантика перемещения(movesemantic)

Наконец, перейдем к практическому применению изложенного выше материала - семантике перемещения. Это одно из самых важных и нужных новшеств, которые привнесли rvalue ссылки в C++(для программистов не занимающихся написанием библиотек общего назначения самое важное, я полагаю). Итак, в чем же оно заключается? Как мы уже выяснили, параметр являющийся rvalue ссылкой находится "на последнем издыхании" и следовательно его ресурсы уже готовы к тому, чтобы быть освобожденными. Как мы можем это использовать, спросите вы? Очень просто - если есть некий объект, чьи ресурсы более не нужны мы

можем забрать(steal) эти ресурсы себе! Не впечатлены? Правильно, необходимо привести пример, чтобы понять что происходит и, главное, зачем это надо:

```
class NodePrivate
{
    friend class Node;

    std::vector<int> m_List1;
    std::vector<int> m_List2;
    std::vector<int> m_List3;
    std::vector<int> m_List4;
    std::vector<int> m_List5;
    std::string m_strVeryLongString;
public:
     NodePrivate(const NodePrivate& Rhs);
}

class Node
{
    std::unique_ptr<NodePrivate> m_spData;
public:
    Node& operator=(const Node& Rhs);
    Node& operator=(Node&& Rhs);
};

NodePrivate::NodePrivate(const NodePrivate& Rhs)
{
    m_List1 = Rhs.m_List1;
    m_List2 = Rhs.m_List2;
    m_List3 = Rhs.m_List3;
    m_List4 = Rhs.m_List4;
    m_List5 = Rhs.m_List5;
    m_strVeryLongString = Rhs.m_strVeryLongString;
}

Node& Node::operator=(const Node& Rhs)
{
    /*
        Тут происходит полное копирование всех внутренних данных
         векторов, строк и т.д. Очень затратная операция.
    */
    m_spData.reset(new NodePrivate(*Rhs.m_spData));
}

Node& Node::operator=(Node&& Rhs)
{
    //Просто перемещаем указатель, никакого копирования!
    m_spData = std::move(Rhs.m_spData);
}
```

```
1
```

Из примера выше можно заметить, что при использование operator= с семантикой перемещения(а это достигается передачей rvalue ссылки качестве параметра) происходить только перемещение указателя(кстати std::unique_ptr это замена std::auto_ptr из предыдущего стандарта, особенностью этого указателя является отсутствие operator= копирования и присутствие operator= перемещения) не выполняется никакого глубокого копирования. Это операция тривиальна и выполняется довольно быстро, чего нельзя сказать о полном копировании всех данных содержащихся в объекте класса NodePrivate происходящем, при использовании operator=(const Node& Rhs).

Кстати, поддержка оператора перемещения добавлена в классы stl и если в вашем классе есть этот оператор то и все stl члены, будут перемещены в результате вызова оператора перемещения! Более того, так как stl знает о семантике перемещения это может дать прирост в производительности при исполнении некоторых операций stl(все мы знаем, что stl очень любит копировать объекты, но с приходом семантики перемещения копирование может быть заменено на перемещение). Это даст ощутимый прирост в производительности с минимальными усилиями с вашей стороны, C++ не зря считается одним из самых "производительных" языков.

С пришествием семантики перемещения мы получили еще одно средство сделать наши программы быстрее. Ведь теперь объекты, которые прекращают свое существование, могут быть использованы без избыточного копирования. Это за нас сделает умный компилятор, а там где компилятор пасует(например если объект не находится на пороге уничтожения, но вы знаете, что в дальнейшем его использования не предвидится) вы можете использовать std::move для перемещения ресурсов, занимаемых этим объектом. Но с этим надо быть осторожным, т.к. вы должны гарантировать, что никто в дальнейшем не будет использовать объект, который был перемещение в противном случае получите "падение" программы.

Можно привести пример, более приближенный к реальности:

```
1  std::vector<char> ReadDataFromSocket(boost::asio::ip::tcp::socket
2  {
3      ...
4      // Надо прочитать крупный массив данных из сокета
5      std::vector<char> Array(BigLength);
6      ...
7      Socket.read_some(boost::asio::buffer(Array));
8      ...
9      // Перемещаем вектор, избавляясь от тяжеловесного копирования
10     return std::move(Array);
11 }
```

*std::move, здесь, используется только для наглядности. Его использование не является обязательным в данном случае.

Элегантно, не правда ли?

Если конструктор перемещения явно не объявлен для класса A, тогда он будет сгенерирован неявно при соблюдении следующих условий(C++11 12.8.9):

> Класс A не содержит явного объявления конструктора копирования
> Класс A не содержит явного объявления оператора копирования
> Класс A не содержит явного объявления оператора перемещения
> Класс A не содержит явного объявления деструктора
> Конструктора копирования не был явно помечен как*deleted*

Если конструктор перемещения не был объявлен явно или не был сгенерирован компилятором, тогда при использовании семантики перемещения будет использован конструктор копирования.

Совершенная передача(perfectforwarding)

Прежде чем описать что же это, такое вернемся к предыдущему стандарту и опишем существующую проблему. Предположим, у нас есть шаблонная функция *foo* принимающая один параметр, и передающая его функции*bar*:

```
template <typename T>
void foo(T& Object)
{
    bar(Object);
}

```

```

```

Итак, все хорошо. Но, что если мы захотим передать, скажем, число 100 в качестве аргумента функции?

Не беда, напишем так:

```
template <typename T>
void foo(const T& Object)
{
    bar(Object);//Ooops
}

```

Но в этом случае bar нарушает const, а следовательно получаем ошибку компиляции. Значит надо предоставить 2 функции bar - константную и нет. А теперь представим, что у функции не один параметр а 2,3, или 5? Получается, что подобная задача очень трудна в реализации, т.к мы имеем (2^n - 1) перегруженных функций, где n - количество аргументов функции. Если вы думаете, что такое количество параметров является плохим стилем и вообще так никто не пишет, тогда обратите свой взор на std::bind, std::make_shared и т.д.

Теперь посмотрим, какое же решение нам предоставляет новый стандарт:

```
template <typename T>
void foo(T&& Object)
{
    bar(std::forward<T>(Object));
}

```

Используя вышеприведённый код проблема с передачей параметров полностью решается, это и называется *совершенной передачей*, т.к. тип аргумента сохраняется между вызовами внешней функции *foo*и внутренней функции *bar*. Больше нет нужды в перегрузке кучи функций - разработчики обобщенного кода могут быть довольны.

Это решение возможно благодаря тому, что если параметром шаблона является T&&, то переданный тип сохранит себя, а std::forward нужен затем, что любой именованный тип внутри функции *foo* превращается в lvalue, а нам нужен исходный тип - для этого и применяется std::forward он сохраняет исходный тип аргумента и лишает его имени(получается T&&), что позволяет в дальнейшем передать его в точности в функцию *bar*.

Почему же T&& сохраняет исходный тип? Это происходит согласно правилам вывода параметров шаблона, которые

а) Исключает ссылки из рассмотрения, и выводят тип согласно переданному аргументу и б) специальному правилу(14.8.2.1/3) касающемуся T&& и lvalue - эта пара на выходе дает T&(на const T&& это правило не распространяется!). Рассмотрим это на примере функции

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

Передаваемый аргумент	Выведенный параметр шаблона	Результирующий тип аргумента функции
lvalue int	int&(см. 14.8.2.1/3)	int & && -> int&
const lvalue int	const int&	const int & && - > const int&
rvalue int	int	int &&
const rvalue int	const int	const int&&

Ну или все тоже самое, только в коде:

```
template<class T>
struct Foo
{
    static void foo()
    {
        std::cout << "foo(): plain" << std::endl;
    }
};

template<>
```

```cpp
struct Foo<int&>
{
    static void foo()
    {
        std::cout << "foo(): int&" << std::endl;
    }
};

template<>
struct Foo<const int&>
{
    static void foo()
    {
        std::cout << "foo(): const int&" << std::endl;
    }
};

template<>
struct Foo<int>
{
    static void foo()
    {
        std::cout << "foo(): int" << std::endl;
    }
};

template<>
struct Foo<const int>
{
    static void foo()
    {
        std::cout << "foo(): const int" << std::endl;
    }
};

int bar()
{
    return 1;
}

const int const_bar()
{
    return 1;
}


template<class T>
void helper(T&&)
{
    Foo<T>::foo();
}

int main()
{

    int i = 1;

    const int j = 1;
    helper(i);
    helper(j);
    helper(bar());
```

```
    helper(const_bar());
}
```

Будет выведено:

```
foo(): int&
foo(): const int&
foo(): int
foo(): const int
```

Что и требовалось доказать.

Вывод

С появлением rvalue ссылок у нас появилась возможность сделать наш код семантически более правильным, а также привнести в него дополнительную скорость за счёт семантики перемещения. Разработчики библиотек получили надежное средство для передачи параметров во внутренние функции без чудовищного количества перегруженных функций. Поэтому rvalue ссылку могут быть по праву признаны одним из самых важных нововведений в ядре языка. Хотя добавление перемещающих operator= и конструктора связаны с нескорыми сложностями(написание нового кода, использование этого кода) я советую всем привыкать писать их, т.к. это сделает ваш код быстрее. Даже если вы явно нигде это не используете, это уже используется в stl и, я уверен, будет использоваться во всех известных библиотеках, которыми занимаются сознательные разработчики, тем более, что rvalue ссылки уже сейчас поддерживаются главными C++ компиляторами (gcc и MSVC). А раз ничего не останавливает нас от использования подобных техник - давайте их использовать!