# Краткое введение в rvalue-ссылки

перевод

C++*, Разработка*, Программирование*

Перевод статьи «A Brief Introduction to Rvalue References», Howard E. Hinnant, Bjarne Stroustrup, Bronek Kozicki.

Rvalue ссылки – маленькое техническое расширение языка C++. Они позволяют программистам избегать логически ненужного копирования и обеспечивать возможность идеальной передачи (perfect forwarding). Прежде всего они предназначены для использования в высоко производительных проектах и библиотеках.

## Введение

Этот документ даёт первичное представление о новой функции языка C++ – rvalue ссылке. Это краткое учебное руководство, а не полная статья. Для получения дополнительной информации посмотрите список ссылок в конце.

## Rvalue ссылка

Rvalue ссылка – это составной тип, очень похожий на традиционную ссылку в C++. Чтобы различать эти два типа, мы будем называть традиционную C++ ссылку lvalue ссылка. Когда будет встречаться термин ссылка, то это относится к обоим видам ссылок, и к lvalue ссылкам, и к rvalue ссылкам.

По семантике lvalue ссылка формируется путём помещая & после некоторого типа.

```
A a;
A& a_ref1 = a;  // это lvalue ссылка
```

Если после некоторого типа поместить &&, то получится rvalue ссылка.

```
A a;
A&& a_ref2 = a;  // это rvalue ссылка
```

Rvalue ссылка ведет себя точно так же, как и lvalue ссылка, за исключением того, что она может быть связана с временным объектом, тогда как lvalue связать с временным (не константным) объектом нельзя.

```
A&  a_ref3 = A();  // Ошибка!
A&& a_ref4 = A();  // Ok
```

Вопрос: С чего бы это могло нам потребоваться?!

Оказывается, что комбинация rvalue ссылок и lvalue ссылок — это то, что необходимо для лёгкой реализации семантики перемещения (move semantics). Rvalue ссылка может также использоваться для достижения идеальной передачи (perfect forwarding), что ранее было нерешенной проблемой в C++. Для большинства программистов rvalue ссылки позволяют создать более производительные библиотеки.

## Семантика перемещений (move semantics)

## Устранение побочных копий

Копирование может быть дорогим удовольствием. К примеру, для двух векторов, когда мы пишем `v2 = v1`, то обычно это вызывает вызов функции, выделение памяти и цикл. Это, конечно, приемлемо, когда нам действительно нужны две копии вектора, но во многих случаях это не так: мы часто копируем вектор из одного места в другое, а потом удаляем старую копию. Рассмотрим:

```
template <class T> swap(T& a, T& b)
{
    T tmp(a);   // сейчас мы имеем две копии объекта a
    a = b;      // теперь у нас есть две копии объекта b
    b = tmp;    // а теперь у нас две копии объекта tmp (т.е. a)
}
```

В действительности нам не нужны копии `a` или `b`, мы просто хотели обменять их. Давайте попробуем еще раз:

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

Этот вызов `move()` возвращает значение объекта, переданного в качестве параметра, но не гарантирует сохранность этого объекта. К примеру, если в качестве параметра в `move()` передать `vector`, то можно обоснованно ожидать, что после работы функции от параметра останется вектор нулевой длины, так как все элементы будут перемещены, а не скопированы. Другими словами, перемещение – это считывание со стиранием (destructive read).

В данном случае мы оптимизировали `swap` специализацией. Однако мы не можем специализировать каждую функцию, которая копирует большой объект непосредственно перед тем, как она удаляет или перезаписывает его. Это было бы неконструктивно.

Главная задача rvalue ссылок состоит в том, чтобы позволить нам реализовывать перемещение без переписывания кода и издержек времени выполнения (runtime overhead).

## Move

Функция `move` в действительности выполняет весьма скромную работу. Её задача состоит в том, чтобы принять либо lvalue, либо rvalue параметр, и вернуть его как rvalue без вызова конструктора копирования:

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

Теперь всё зависит от клиентского кода, где должны быть перегружены ключевые функции (например,

конструктор копирования и оператор присваивания), определяющие будет ли параметр lvalue или rvalue. Если параметр lvalue, то необходимо выполнить копирование. Если rvalue, то можно безопасно выполнить перемещение.

### Перегрузка для lvalue/rvalue

Рассмотрим простой класс, который владеет ресурсом и также обеспечивает семантику копирования (конструктор копирования и оператор присваивания). Например, `clone_ptr` мог бы владеть указателем и вызвать у него дорогой метод`clone()` для копирования:

```
template <class T>
class clone_ptr
{
private:
    T* ptr;
public:
    // Конструктор
    explicit clone_ptr(T* p = 0) : ptr(p) {}

    // Деструктор
    ~clone_ptr() {delete ptr;}

    // Семантика копирования
    clone_ptr(const clone_ptr& p)
        : ptr(p.ptr ? p.ptr->clone() : 0) {}

    clone_ptr& operator=(const clone_ptr& p)
    {
        if (this != &p)
        {
            delete ptr;
            ptr = p.ptr ? p.ptr->clone() : 0;
        }
        return *this;
    }

    // Семантика перемещения
    clone_ptr(clone_ptr&& p)
        : ptr(p.ptr) {p.ptr = 0;}

    clone_ptr& operator=(clone_ptr&& p)
    {
        std::swap(ptr, p.ptr);
        return *this;
    }

    // Прочие операции
    T& operator*() const {return *ptr;}
    // ...
};
```

За исключением семантики перемещения, `clone_ptr` – это код, который можно найти в сегодняшних

книгах по C++. Пользователи могли бы использовать`clone_ptr` так:

```
clone_ptr<base> p1(new derived);
// ...
clone_ptr<base> p2 = p1;  // и p2 и p1 владеют каждый своим собственным указателем
```

Обратите внимание, что выполнение конструктора копирования или оператора присвоения для `clone_ptr` являются относительно дорогой операцией. Однако, когда источник копии является rvalue, можно избежать вызова потенциально дорогой операции `clone()`, воруя указатель rvalue (никто не заметит!). В семантике перемещения конструктор перемещения оставляет значение rvalue в создаваемом объекте, а оператор присваивания меняет местами значения текущего объекта с объектом rvalue ссылки.

Теперь, когда код пытается скопировать rvalue `clone_ptr`, или если есть явное разрешение считать источник копии rvalue (используя `std::move`), работа выполнится намного быстрее.

```
clone_ptr<base> p1(new derived);
// ...
clone_ptr<base> p2 = std::move(p1); // теперь p2 владеет ссылкой, вместо p1
```

Для классов, составленных из других классов (или через включение, или через наследование), конструктор перемещения и перемещающее присвоение может легко быть реализовано при использовании функции `std::move`.

```
class Derived
    : public Base
{
    std::vector<int> vec;
    std::string name;
    // ...
public:
    // ...
    // Семантика перемещения
    Derived(Derived&& x)               // объявлен как rvalue
        : Base(std::move(x)),
          vec(std::move(x.vec)),
          name(std::move(x.name)) { }

    Derived& operator=(Derived&& x)    // объявлен как rvalue
    {
        Base::operator=(std::move(x));
        vec  = std::move(x.vec);
        name = std::move(x.name);
        return *this;
    }
    // ...
};
```

Каждый подобъект будет теперь обработан как rvalue в конструкторе перемещения и операторе перемещающего присваивания объекта. У `std::vector` и `std::string`операции перемещения уже

реализованы (точно так же, как и у нашего`clone_ptr`), которые позволяют избежать значительно более дорогих операций копирования.

Стоит отметить, что параметр `x` обработан как lvalue в операциях перемещения, несмотря на то, что он объявлен как rvalue ссылка. Поэтому необходимо использовать `move(x)` вместо просто `x` при передаче базовому классу. Это ключевой механизм безопасности семантики перемещения, разработанной для предотвращения случайной попытки двойного перемещения из некоторой именованной переменной. Все перемещения происходят только из rvalues или с явным приведением к rvalue (при помощи `std::move`). Если у переменной есть имя, то это lvalue.

Вопрос: А как насчет типов, которые не владеют ресурсами? (Например,`std::complex`?)

В этом случае не требуется проводить никакой работы. Конструктор копирования уже оптимален для копирования с rvalue.

## Перемещаемые, но не копируемые типы

К некоторым типам семантика копирования не применима, но их можно перемещать. Например:

- `fstream`
- `unique_ptr` (не разделяемое и не копируемое владение)
- Тип, представляющий поток выполнения

Если такие типы делать перемещаемыми (хотя они остаются не копируемыми), то удобство их использования чрезвычайно увеличивается. Перемещаемый, но не копируемый объект может быть возвращен по значению из фабричного метода (паттерн):

```
ifstream find_and_open_data_file(/* ... */);
...
ifstream data_file = find_and_open_data_file(/* ... */);  // Никаких копий!
```

В этом примере базовый дескриптор файла передан из одного объекта в другой, т.к. источник `ifstream` является rvalue. В любом момент времени есть только один дескриптор файла, и только один `ifstream` владеет им.

Перемещаемый, но не копируемый тип также может быть помещён в стандартные контейнеры. Если контейнеру необходимо “скопировать” элемент внутри себя (например, при реалокации `vector`), он просто переместит его вместо копирования.

```
vector<unique_ptr<base>> v1, v2;
v1.push_back(unique_ptr<base>(new derived()));  // OK, перемещение без копирования
...
v2 = v1;              // Ошибка времени компиляции! Это не копируемый тип.
v2 = move(v1);        // Нормальное перемещение. Владение указателем будет передано v2.
```

Многие стандартные алгоритмы извлекают выгоду от перемещения элементов последовательности вместо их копирования. Это не только обеспечивает лучшую производительность (как в случае `std::swap`, реализация которого описала выше), но и позволяет этим алгоритмам работать с некопируемыми (но перемещаемыми) типами. Например, следующий код сортирует`vector<unique_ptr<T>>`, основываясь на типе, который хранится в умном указателе:

```
struct indirect_less
{
    template <class T>
    bool operator()(const T& x, const T& y)
        {return *x < *y;}
};
...
std::vector<std::unique_ptr<A>> v;
...
std::sort(v.begin(), v.end(), indirect_less());
```

Поскольку алгоритм сортировки перемещает объекты `unique_ptr`, он будет использовать `swap` (который больше не требует поддержки копируемости от объектов, значения которых он обменивает) или конструктор перемещения / оператор перемещающего присваивания. Таким образом, на протяжении всей работы алгоритма поддерживается инвариант, по которому каждый хранимый объект находится во владении только одного умного указателя. Если бы алгоритм предпринял попытку копирования (к примеру, по ошибке программиста), то результатом была бы ошибка времени компиляции.

## Идеальная передача (perfect forwarding)

Рассмотрим универсальный фабричный метод, который возвращает`std::shared_ptr` для только что созданного универсального типа. Такие фабричные методы ценны для инкапсуляции и локализации выделения ресурсов. Очевидно, фабричный метод должен принимать точно такой же набор параметров, что и конструктор типа создаваемого объекта. Сейчас это может быть реализовано так:

```
template <class T>
std::shared_ptr<T>
factory()   // версия без аргументов
{
    return std::shared_ptr<T>(new T);
}

template <class T, class A1>
std::shared_ptr<T>
factory(const A1& a1)   // версия с одним аргументом
{
    return std::shared_ptr<T>(new T(a1));
}

// все остальные версии
```

В интересах краткости мы будем фокусироваться на простой версии с одним параметром. Например:

```
std::shared_ptr<A> p = factory<A>(5);
```

Вопрос: Что будет, если конструктор `T` получает параметр по не константной ссылке?

В этом случае мы получаем ошибку времени компиляции, поскольку константный параметр функции `factory` не будет связываться с неконстантным параметром конструктора типа `T`.

Для решения этой проблемы можно использовать неконстантный параметр в функции `factory`:

```
template <class T, class A1>
std::shared_ptr<T>
factory(A1& a1)
{
    return std::shared_ptr<T>(new T(a1));
}
```

Так намного лучше. Если тип с модификатором `const` будет передан `factory`, то константа будет выведена в шаблонный параметр (например, `A1`) и затем должным образом передана конструктору `T`. Точно так же, если фабрике будет передан неконстантный параметр, то он будет правильно передан конструктору `T` как неконстанта. В действительности именно так чаще всего реализуется передача параметра (например, `std::bind`).

Теперь рассмотрим следующую ситуацию:

```
std::shared_ptr<A> p = factory<A>(5);    // Ошибка!
A* q = new A(5);                                    // OK
```

Этот пример работал с первой версией `factory`, но теперь аргумент «5» вызывает шаблон `factory`, который будет выведен как `int&` и впоследствии не сможет быть связанным с rvalue «5». Таким образом, ни одно решение нельзя считать правильным, каждый страдает своими проблемами.

Вопрос: А может сделать перегрузку для каждой комбинации `AI`& и `const AI`&?

Это позволило бы нам обрабатывать все примеры, но приведёт к экспоненциальной стоимости: для нашего случая с двумя параметрами это потребовало бы 4 перегрузки. Для фабрики с тремя параметрами мы нуждались бы в 8 дополнительных перегрузках. Для фабрики с четырьмя параметрами потребовалось бы уже 16 перегрузок и т.д. Это совершенно не масштабируемое решение.

Rvalue ссылки предлагают простое и масштабируемое решение этой задачи:

```
template <class T, class A1>
std::shared_ptr<T>
factory(A1&& a1)
{
    return std::shared_ptr<T>(new T(std::forward<A1>(a1)));
}
```

Теперь rvalue параметры могут быть связаны с параметрами `factory`. Если параметр `const`, то он будет выведен в шаблонный тип параметра `factory`.

Вопрос: Что за функция `forward` используется в этом решении?

Как и `move`, `forward` — это простая стандартная библиотечная функция, используемая, чтобы выразить намерение непосредственно и явно, а не посредством потенциально загадочного использования ссылок. Мы хотим передать параметр `a1`, и просто заявляем об этом.

Здесь, `forward` сохраняет lvalue/rvalue параметр, который был передан `factory`. Если `factory` был передан rvalue, то при помощи `forward` и конструктору `T` будет передан rvalue. Точно так же, если

lvalue параметр передан `factory`, он же будет передан конструктору `T` как lvalue.

Определение функции `forward` может выглядеть примерно так:

```
template <class T>
struct identity
{
    typedef T type;
};

template <class T>
T&& forward(typename identity<T>::type&& a)
{
    return a;
}
```

## Ссылки

Поскольку одна из основных целей этой заметки краткость, некоторые детали были сознательно опущены. Тем не менее, здесь покрыто 95% знаний по этой теме.

Для получения дальнейшей информации по семантики перемещения, такой как тесты производительности, детали перемещений, не копируемые типы, и многое другое смотрите N1377.

Для полной информации об обработке проблемы передачи смотрите N1385.

Для дальнейшего изучения rvalue ссылок (помимо семантики перемещения и идеальной передачи), смотрите N1690.

Для формулировок изменений языка, требуемых rvalue ссылками, смотрите N1952.

Обзор по rvalue ссылкам и семантике перемещения, смотрите N2027.

Сводка по всем влияниям rvalue ссылок на стандартную библиотеке, находится вN1771.

Предложения и формулировки для изменений в библиотеке, требующих использовать в rvalue ссылку, смотрите:

- N1856
- N1857
- N1858
- N1859
- N1860
- N1861
- N1862

Предложения распостранить rvalue ссылку на неявный объектный параметр (this), смотрите N1821.