

Декларатор ссылки rvalue:&&


04.11.2016 • Время чтения: 10 мин • 

В этой статье

- [Синтаксис](#)
- [Примечания](#)
- [Семантика перемещения](#)
- [Точная пересылка](#)
- [Дополнительные свойства ссылок rvalue](#)
- [Сводка](#)
- [См. также](#)

Содержит ссылку на выражение rvalue.

Синтаксис

	 Копировать
type-id && cast-expression	

Примечания

Ссылки rvalue позволяют различать значения lvalue и rvalue. Ссылки lvalue и rvalue синтаксически и семантически аналогичны, однако они подчиняются несколько различающимся правилам. Дополнительные сведения о значения lvalue и rvalue см. в разделе [значения lvalue и rvalues](#). Дополнительные сведения о ссылках lvalue см. в разделе [декларатор ссылки lvalue: &](#).

В следующих разделах описывается, как ссылки rvalue поддерживают реализацию *семантики перемещения* и *идеальной пересылки*.


Семантика перемещения

Ссылки rvalue поддерживают реализацию семантики *перемещения*, что может значительно повысить производительность приложений. Семантика перемещения позволяет создавать код, который переносит ресурсы (например, динамически выделяемую память) из одного объекта в другой. Семантика перемещения работает, поскольку она позволяет переносить ресурсы из временных объектов, на которые невозможно сослаться из других мест в программе.

Для реализации семантики перемещения, как правило, предоставляется *конструктор перемещения* и при необходимости оператор присваивания перемещения (**operator =**) к

вашему классу. Операции копирования и присваивания, источниками которых являются значения rvalue, затем автоматически используют семантику перемещения. В отличие от конструктора копирования по умолчанию, компилятор не предоставляет конструктор перемещения по умолчанию. Дополнительные сведения о написании конструктора перемещения и его использовании в приложении см. в разделе [конструкторы перемещения и операторы присваивания перемещения C++\(\)](#).

Можно также перегрузить обычные функции и операторы, чтобы воспользоваться преимуществами семантики перемещения. В Visual Studio 2010 введена семантика перемещения C++ в стандартную библиотеку. Например, класс `string` реализует операции, использующие семантику перемещения. Рассмотрим следующий пример, в котором объединяются несколько строк и выводится результат:

C++	 Копировать
<pre>// string_concatenation.cpp // compile with: /EHsc #include <iostream> #include <string> using namespace std; int main() { string s = string("h") + "e" + "ll" + "o"; cout << s << endl; }</pre>	

До выхода Visual Studio 2010 каждый вызов **оператор +** выделяет и возвращает новый временный `string` объект (rvalue). **оператор +** не может добавить одну строку в другую, так как не знает, являются ли исходные строки значения lvalue или rvalue. Если обе исходные строки являются значениями lvalue, на них могут указывать ссылки из какого-либо другого места программы, поэтому их не следует изменять. При использовании ссылок rvalue **оператор +** может быть изменен для получения rvalue, на который нельзя ссылаться в других местах программы. Таким образом, **оператор +** теперь может добавлять одну строку к другой. Это может значительно снизить количество операций динамического выделения памяти, которые должен выполнять класс `string`. Дополнительные сведения о классе см `string` . в разделе [класс basic_string](#).

Семантика перемещения также помогает, когда компилятор не может использовать оптимизацию возвращаемого значения (RVO) или оптимизацию именованного возвращаемого значения (NRVO). В таких случаях компилятор вызывает конструктор перемещения, если он определен в типе. Дополнительные сведения об оптимизации именованных возвращаемых значений см. [в разделе Оптимизация возвращаемого значения с именем в Visual Studio 2005](#).

Для лучшего понимания семантики перемещения рассмотрим пример вставки элемента в объект `vector` . Если ресурсы объекта `vector` превышены, объект `vector` должен заново выделить память для своих элементов, а затем скопировать каждый элемент в другое расположение в памяти, чтобы освободить место для добавленного элемента. Когда операция вставки копирует элемент, она создает новый элемент, вызывает конструктор копирования для копирования данных из предыдущего элемента в новый элемент, а затем уничтожает предыдущий элемент. Семантика перемещения позволяет перемещать объекты напрямую, не выполняя ресурсоемкие операции выделения памяти и копирования.

Чтобы воспользоваться преимуществами семантики перемещения в примере `vector` , можно создать конструктор перемещения для перемещения данных из одного объекта в другой.

Дополнительные сведения о поведении семантики перемещения в C++ стандартную библиотеку в Visual Studio 2010 см. в разделе [C++ Стандартная библиотека](#).

Точная пересылка

Точная пересылка уменьшает необходимость в использовании перегруженных функций и позволяет избежать проблем пересылки. *Проблема пересылки* может возникнуть при написании универсальной функции, которая принимает ссылки в качестве параметров и передает (или *пересылает*) эти параметры другой функции. Например, если универсальная функция принимает параметр типа `const T&`, вызываемая функция не может изменять значение этого параметра. Если универсальная функция принимает параметр типа `T&`, эта функция не может вызываться с использованием значения rvalue (такого как временный объект или целочисленный литерал).

Обычно для решения этой проблемы необходимо предоставить перегруженные версии универсальной функции, принимающие для каждого из своих параметров значения `T&` и `const T&`. В результате число перегруженных функций экспоненциально возрастает по мере увеличения числа параметров. Ссылки rvalue позволяют создать одну версию функции, принимающую произвольные аргументы и пересылающую их в другую функцию, как если бы эта другая функция вызывалась напрямую.

Рассмотрим следующий пример, в котором определяются четыре типа: `w`, `x`, `y` и `z`. Конструктор для каждого типа в качестве параметров принимает Разное сочетание константных и неконстантных ссылок lvalue.

C++

Копировать

```
struct w
{
    w(int&, int&) {}
};

struct x
{
    x(const int&, int&) {}
};

struct y
{
    y(int&, const int&) {}
};

struct z
{
    z(const int&, const int&) {}
};
```

Предположим, требуется написать универсальную функцию, которая создает объекты. В следующем примере показан один из возможных способов написания этой функции:

C++

Копировать

```
template <typename T, typename A1, typename A2>
T* factory(A1& a1, A2& a2)
{
    return new T(a1, a2);
}
```

В следующем примере показан допустимый вызов функции `factory`:

C++	Копировать
<pre>int a = 4, b = 5; W* pw = factory<W>(a, b);</pre>	

Однако следующий пример содержит недопустимый вызов функции `factory`, поскольку функция `factory` принимает в качестве параметров ссылки lvalue, допускающие изменения, но вызывается с использованием значений rvalue:

C++	Копировать
<pre>Z* pz = factory<Z>(2, 2);</pre>	

Обычно для решения этой проблемы необходимо создать перегруженные версии функции `factory` для каждого сочетания параметров `A&` и `const A&`. Ссылки rvalue позволяют создать одну версию функции `factory`, как показано в следующем примере:

C++	Копировать
<pre>template <typename T, typename A1, typename A2> T* factory(A1&& a1, A2&& a2) { return new T(std::forward<A1>(a1), std::forward<A2>(a2)); }</pre>	

В этом примере в качестве параметров функции `factory` используются ссылки rvalue. Функция [std::Forward](#) пересылает параметры функции `Factory` конструктору класса шаблона.

В следующем примере показана функция `main`, использующая измененную функцию `factory` для создания экземпляров классов `w`, `x`, `y` и `z`. Измененная функция `factory` пересылает свои параметры (значения lvalue или rvalue) в конструктор соответствующего класса.

C++	Копировать
<pre>int main() { int a = 4, b = 5;</pre>	

```
W* pw = factory<W>(a, b);
X* px = factory<X>(2, b);
Y* py = factory<Y>(a, 2);
Z* pz = factory<Z>(2, 2);

delete pw;
delete px;
delete py;
delete pz;
}
```

Дополнительные свойства ссылок rvalue

Можно перегрузить функцию, чтобы получить ссылку lvalue и ссылку rvalue.

Перегружая функцию для получения константной ссылки lvalue или ссылки rvalue, можно написать код, который различает неизменяемые объекты (значения lvalue) и изменяемые временные значения (rvalue). Вы можете передать объект в функцию, которая принимает ссылку rvalue, если только объект не помечен как **const**. В следующем примере показана перегруженная функция `f`, принимающая ссылки lvalue и rvalue. Функция `main` вызывает функцию `f` как со значениями lvalue, так и со значениями rvalue.

C++Копировать

```
// reference-overload.cpp
// Compile with: /EHsc
#include <iostream>
using namespace std;

// A class that contains a memory resource.
class MemoryBlock
{
    // TODO: Add resources for the class here.
};

void f(const MemoryBlock&)
{
    cout << "In f(const MemoryBlock&). This version cannot modify the parameter." << endl;
}


void f(MemoryBlock&&)
{
    cout << "In f(MemoryBlock&&). This version can modify the parameter." << endl;
}

int main()
{
    MemoryBlock block;
    f(block);
    f(MemoryBlock());
}
```

```
}

```

В этом примере выводятся следующие данные:


Output	 Копировать
In f(const MemoryBlock&). This version cannot modify the parameter. In f(MemoryBlock&&). This version can modify the parameter.	

В этом примере при первом вызове функции `f` в качестве аргумента передается локальная переменная (значение lvalue). При втором вызове функции `f` в качестве аргумента передается временный объект. Поскольку на временный объект невозможно сослаться из какого-либо другого места в программе, вызов привязывается к перегруженной версии функции `f`, которая принимает ссылку rvalue и может свободно изменять этот объект.

Компилятор рассматривает именованную ссылку rvalue как lvalue и неименованную ссылку rvalue в качестве rvalue.

При создании функции, которая принимает в качестве параметра ссылку rvalue, в теле функции этот параметр обрабатывается как значение lvalue. Компилятор обрабатывает именованную ссылку rvalue как значение lvalue, поскольку на именованный объект возможны ссылки из нескольких частей программы; было бы опасно разрешить нескольким частям программы изменять или удалять ресурсы из этого объекта. Например, если несколько частей программы попытаются переместить ресурсы из одного и того же объекта, только первая часть сможет успешно переместить ресурс.

В следующем примере показана перегруженная функция `g`, принимающая ссылки lvalue и rvalue. Функция `f` принимает ссылку rvalue в качестве своего параметра (именованная ссылка rvalue) и возвращает ссылку rvalue (безымянная ссылка rvalue). При вызове функции `g` из функции `f` механизм разрешения перегрузок выбирает версию `g`, которая принимает ссылку lvalue, так как в теле функции `f` ее параметр рассматривается как значение lvalue. При вызове функции `g` из функции `main` механизм разрешения перегрузок выбирает версию `g`, которая принимает ссылку rvalue, так как функция `f` возвращает ссылку rvalue.


C++	 Копировать
<pre>// named-reference.cpp // Compile with: /EHsc #include <iostream> using namespace std; // A class that contains a memory resource. class MemoryBlock { // TODO: Add resources for the class here. }; void g(const MemoryBlock&) { cout << "In g(const MemoryBlock&)." << endl; } void g(MemoryBlock&&)</pre>	

```
{
    cout << "In g(MemoryBlock&&)." << endl;
}

MemoryBlock&& f(MemoryBlock&& block)
{
    g(block);
    return move(block);
}

int main()
{
    g(f(MemoryBlock()));
}
```


В этом примере выводятся следующие данные:

C++	 Копировать
In g(const MemoryBlock&). In g(MemoryBlock&&).	

В этом примере функция `main` передает значение `rvalue` в функцию `f`. В теле функции `f` ее именованный параметр рассматривается как значение `lvalue`. При вызове функции `f` из функции `g` параметр связывается со ссылкой `lvalue` (первая перегруженная версия функции `g`).

- Можно привести `lvalue` к ссылке `rvalue`.

Функция C++ стандартной библиотеки [std::Move](#) позволяет преобразовать объект в ссылку `rvalue` на этот объект. Кроме того, можно использовать ключевое слово **static_cast** для приведения левостороннего значения к ссылке `rvalue`, как показано в следующем примере:

C++	 Копировать
<pre>// cast-reference.cpp // Compile with: /EHsc #include <iostream> using namespace std; // A class that contains a memory resource. class MemoryBlock { // TODO: Add resources for the class here. }; void g(const MemoryBlock&) { cout << "In g(const MemoryBlock&)." << endl; }</pre>	

```
void g(MemoryBlock&&)
{
    cout << "In g(MemoryBlock&&)." << endl;
}

int main()
{
    MemoryBlock block;
    g(block);
    g(static_cast<MemoryBlock&&>(block));
}
```

В этом примере выводятся следующие данные:

C++	Копировать
In g(const MemoryBlock&). In g(MemoryBlock&&).	

Шаблоны функций выводят типы аргументов шаблонов, а затем используют правила свертывания ссылок.

Часто создается шаблон функции, который передает (или пересылает)свои параметры другой функции. Важно понимать, как работает вывод типа шаблона для шаблонов функций, принимающих ссылки rvalue.

Если аргумент функции является значением rvalue, компилятор считает, что аргумент является ссылкой rvalue. Например, при передаче ссылки rvalue на объект типа x в шаблон функции, который принимает в качестве параметра тип T&&, логика вывода шаблона определит, что T — это x. Поэтому параметр имеет тип x&&. Если аргумент функции является lvalue или **const** lvalue, компилятор выводит свой тип в виде ссылки lvalue или **const** левостороннего ссылки этого типа.

В следующем примере объявляется один шаблон структуры, который затем специализируется для различных ссылочных типов. Функция print_type_and_value принимает в качестве параметра ссылку rvalue и пересылает ее в соответствующую специализированную версию метода S::print. Функция main показывает различные способы вызова метода S::print.

```
C++
Копировать

// template-type-deduction.cpp
// Compile with: /EHsc
#include <iostream>
#include <string>
using namespace std;

template<typename T> struct S;

// The following structures specialize S by
// lvalue reference (T&), const lvalue reference (const T&),
```



```
// rvalue reference (T&&), and const rvalue reference (const T&&).
// Each structure provides a print method that prints the type of
// the structure and its parameter.

template<typename T> struct S<T&> {
    static void print(T& t)
    {
        cout << "print<T&>: " << t << endl;
    }
};

template<typename T> struct S<const T&> {
    static void print(const T& t)
    {
        cout << "print<const T&>: " << t << endl;
    }
};

template<typename T> struct S<T&&> {
    static void print(T&& t)
    {
        cout << "print<T&&>: " << t << endl;
    }
};

template<typename T> struct S<const T&&> {
    static void print(const T&& t)
    {
        cout << "print<const T&&>: " << t << endl;
    }
};

// This function forwards its parameter to a specialized
// version of the S type.
template <typename T> void print_type_and_value(T&& t)
{
    S<T&&>::print(std::forward<T>(t));
}

// This function returns the constant string "fourth".
const string fourth() { return string("fourth"); }

int main()
{
    // The following call resolves to:
    // print_type_and_value<string&>(string& && t)
    // Which collapses to:
    // print_type_and_value<string&>(string& t)
    string s1("first");
    print_type_and_value(s1);
}
```

```
// The following call resolves to:  
// print_type_and_value<const string&>(const string& && t)  
// Which collapses to:  
// print_type_and_value<const string&>(const string& t)  
const string s2("second");  
print_type_and_value(s2);  
  
// The following call resolves to:  
// print_type_and_value<string&&>(string&& t)  
print_type_and_value(string("third"));  
  
// The following call resolves to:  
// print_type_and_value<const string&&>(const string&& t)  
print_type_and_value(fourth());  
}
```

В этом примере выводятся следующие данные:

C++	Копировать
print<T&>: first print<const T&>: second print<T&&>: third print<const T&&>: fourth	

Чтобы разрешить каждый вызов функции `print_type_and_value`, компилятор сначала выполняет выводение аргументов шаблона. Затем при подстановке аргументов шаблона, выведенных для типов параметра, компилятор применяет правила сворачивания ссылок. Например, при передаче локальной переменной `s1` в функцию `print_type_and_value` компилятор создает следующую сигнатуру функции:

C++	Копировать
print_type_and_value<string&>(string& && t)	

Используя правила сворачивания ссылок, компилятор уменьшает сигнатуру до следующей:

C++	Копировать
print_type_and_value<string&>(string& t)	

Затем эта версия функции `print_type_and_value` пересылает свой параметр в требуемую специализированную версию метода `S::print`.

В следующей таблице приведены правила сворачивания ссылок для вывода типа аргументов шаблонов:

Развернутый тип	Свернутый тип
T& &	T&
T& &&	T&
T&& &	T&
T&& &&	T&&

Вывод аргументов шаблонов — это важный элемент реализации точной пересылки. Более подробно точная пересылка рассматривается выше в подразделе "Точная пересылка" этого раздела.

Сводка

Ссылки rvalue различают значения lvalue и rvalue. Они помогают повысить производительность приложений, устраняя необходимость в ненужных операциях выделения памяти и копирования. Они также позволяют создавать одну версию функции, принимающую произвольные аргументы и пересылающую их в другую функцию, как если бы эта другая функция вызывалась напрямую.

См. также

- [Выражения с унарными операторами](#)
- [Декларатор ссылки Lvalue: &](#)
- [Значения Lvalues и Rvalues](#)
- [Конструкторы перемещения и операторы присваивания перемещением \(C++ \)](#)
- [Стандартная библиотека C++](#)

Были ли сведения на этой странице полезными?

 Да  Нет