

Семантика перемещения I



Категории значений (value categories)

Выражение (expression)

- *Выражение* - последовательность операций и их операндов, задающая некоторое вычисление.
- Результат вычисления проявляется в возвращаемом значении (`x + y`) и/или в форме "побочного эффекта" (`x++`).
- Каждое выражение характеризуется **типом возвращаемого значения** и **категорией значения**.

<code>int x = 0;</code>	<code>//</code>	тип		побочный эффект		категория значения
<code>int y = 1;</code>	<code>//</code>	-----		-----		-----
<code>x + y;</code>	<code>//</code>	<code>int</code>		нет		???
<code>++x;</code>	<code>//</code>	<code>int</code>		увеличение x на 1		???
<code>x++;</code>	<code>//</code>	<code>int</code>		увеличение x на 1		???
<code>new int;</code>	<code>//</code>	<code>int*</code>		выделение ресурса		???
<code>delete p;</code>	<code>//</code>	<code>void</code>		освобождение ресурса		???

Value categories

Категория значения - вторая характеристика выражения в C++ (первая - тип).

Главный вопрос, на который отвечает категория значения:

- материален ли результат этого выражения, то есть существует ли он в виде объекта в памяти или нет? (идентичность)

<code>int x = 0;</code>	<code>// тип</code>	<code> </code>	<code>побочный эффект</code>	<code> </code>	<code>категория значения</code>
<code>int y = 1;</code>	<code>//</code>	<code>-----</code>			
<code>x + y;</code>	<code>// int</code>	<code> </code>	<code>нет</code>	<code> </code>	<code>??? (не материален)</code>
<code>++x;</code>	<code>// int</code>	<code> </code>	<code>увеличение x на 1</code>	<code> </code>	<code>??? (материален - x)</code>
<code>x++;</code>	<code>// int</code>	<code> </code>	<code>увеличение x на 1</code>	<code> </code>	<code>??? (не материален)</code>
<code>new int;</code>	<code>// int*</code>	<code> </code>	<code>выделение ресурса</code>	<code> </code>	<code>??? (не материален)</code>
<code>delete p;</code>	<code>// void</code>	<code> </code>	<code>освобождение ресурса</code>	<code> </code>	<code>??? (не материален)</code>

lvalue

lvalue - категория значений, которая обладает идентичностью (но не перемещаемая).

Неформально: к *lvalue* относится все, у чего есть постоянное место в памяти (прописка). Критерий - у выражения можно получить адрес.

- Переменная - всегда *lvalue*!
- Результат функции/операции, возвращающей ссылку
- Строковый литерал (относится к массиву, который содержит символы)

```
x;           // lvalue, &x - валидная операция
++x;         // lvalue, &++x - валидная операция
"abc";       // lvalue, &"abc" - валидная операция
x + 1;       // not lvalue, &(x + 1) - не валидная операция
```

Факт дня: раньше lvalue назывались выражения, которые могли стоять слева от оператора присваивания, отсюда и название - l(ef)t-value

lvalue: конкурс!

Укажите тип и категорию значения выражения.

```
int x = 0;  
int& rx = x;  
const int& crx = x;  
const int& tmp = 11;  
int f();  
int& g();
```

```
x;      /* ??? */  
5;      /* ??? */  
&x;     /* ??? */  
rx;     /* ??? */  
crx;    /* ??? */  
tmp;    /* ??? */  
f();    /* ??? */  
f() + f(); /* ??? */
```

```
g;      // ???  
&f;     // ???  
g();    // ???  
++x;    // ???  
x++;    // ???  
*(&x + 1); // ???  
"lvalue"; // ???  
g() + g(); // ???
```

lvalue: конкурс!

Укажите тип и категорию значения выражения.

```
int x = 0;  
int& rx = x;  
const int& crx = x;  
const int& tmp = 11;  
int f();  
int& g();
```

x;	/* тип - int, lvalue	*/ g;	// тип - int&(), lvalue
5;	/* тип - int, не lvalue	*/ f;	// тип - int(), lvalue
&x;	/* тип - int*, не lvalue	*/ g();	// тип - int, lvalue
rx;	/* тип - int, lvalue	*/ ++x;	// тип - int, lvalue
crx;	/* тип - const int, lvalue	*/ x++;	// тип - int, не lvalue
tmp;	/* тип - const int, lvalue	*/ *(&x + 1);	// тип - int, lvalue
f();	/* тип - int, не lvalue	*/ "lvalue";	// тип - const char[7], lval
f() + f();	/* тип - int, не lvalue	*/ g() + g();	// тип - int, не lvalue

rvalue

rvalue - перемещаемая категория значений. Делится на *prvalue* (pure rvalue) - значение без идентичности и *xvalue* (expired value) - значение с идентичностью.

Неформально:

К *rvalue* относится все, что не относится к *lvalue* (временные значения).

- Литералы (кроме строкового)
- Результат функции/операции, возвращающей значение
- `this`
- Значение типа перечисления (`enum`)
- Параметр шаблона не являющийся типом (если это не ссылка)

Виды ссылок

lvalue ссылки

lvalue-ссылка - это ссылка, которая может связываться с результатом *lvalue* выражения. Эта ссылка становится псевдонимом объекта, на который она ссылается.

```
int& f();  
int x = 0;  
  
int& rx = x;           // lvalue ссылка на x  
const int& crx = x;    // const lvalue ссылка на x  
int& ref = f();        // lvalue ссылка на результат f()  
int& rx2 = ++x;        // lvalue ссылка на x  
  
int& y = 0;            // CE  
int& z = x++;          // CE
```

Исключение - константная *lvalue* ссылка (продлевает жизнь временного объекта)

```
const int& tmp = 11;    // 11 располагается в области памяти с именем tmp
```

rvalue ссылки (C++11)

rvalue-ссылка - это ссылка, которая может связываться с результатом *rvalue* выражения. Эта ссылка продлевает жизнь объекта и в отличие от константных lvalue ссылок позволяет изменять объект.

```
int f();  
int x = 0;  
  
int&& rry = 11;    // rvalue ссылка, инициализируется 11  
rry = 12;         // ok  
int&& rrz = f();   // ok  
int&& rrt = x++;   // ok  
  
int&& y = x;       // CE  
int&& z = ++x;     // CE  
  
int& ref = rry;   // ???
```

rvalue ссылки (C++11)

rvalue-ссылка - это ссылка, которая может связываться с результатом *rvalue* выражения. Эта ссылка продлевает жизнь объекта и в отличие от константных *lvalue* ссылок позволяет изменять объект.

```
int f();

int x = 0;
int&& rry = 11;    // rvalue ссылка, инициализируется 11
rry = 12;         // Ok
int&& rrz = f();    // Ok
int&& rrt = x++;    // Ok

int&& y = x;        // CE
int&& z = ++x;      // CE
```

```
int& ref = rry;    // Ok: вы же помните, что переменная - ВСЕГДА lvalue?
```

rvalue ссылки (C++11)

С: Верно ли, что в предыдущих "Ok" примерах можно было заменить `int&&` на `int` и все работало бы так же?

П: Да.

С: Получается, что rvalue ссылки не нужны?

П: ...



Перегрузка функций по виду ссылки

Мотивация: AddressOf

Хотим написать функцию получения адреса объекта

```
template <class T>
const T* AddressOf(const T& value) {
    return &value;
}

// ...

int x = 0;
auto array = new int[10];

AddressOf(x);           // == &x
AddressOf(++x);         // == &x
AddressOf(array[5]);    // == array + 5
AddressOf(10);          // ?
```

В чем проблема?

Мотивация `AddressOf`

```
template <class T>
const T* AddressOf(const T& value) {
    return &value;
}

// ...

auto ptr = AddressOf(10); // Возвращает "висячий указатель"!
*ptr; // Undefined Behaviour
```

В этом случае `AddressOf` возвращает адрес локальной переменной `value`, которая уничтожится при выходе из функции!

Мотивация: ссылочное поле класса

Хотим сохранить ссылку на внешнюю переменную в поле класса:

```
class A {  
    const int& cref_;  
  
public:  
    A(const int& value) : cref_(value) {}  
    const int& GetRef() { return cref_; }  
    const int* GetPtr() { return &cref_; }  
};  
  
int x = 0;  
auto array = new int[10];  
  
A a(x);  
A b(array[5]);  
A c(0);
```

Что здесь не так?

Мотивация: ссылочное поле класса

```
class A {  
    const int& cref_;  
  
public:  
    A(const int& value) : cref_(value) {}  
    const int& GetRef() { return cref_; }  
    const int* GetPtr() { return &cref_; }  
};  
  
A c(0);  
c.GetRef(); // Undefined Behaviour
```

0 связывается с локальной ссылкой `value`, с ней связывается ссылка `cref_`, а затем `value` уничтожается. Теперь `cref_` - висячая ссылка, вы восхитительны.

Решение

В обоих примерах выше мы не смогли отличить *lvalue* от *rvalue* при передаче значения в функцию.

А что поможет нам разделять аргументы на временные (*rvalue*) и не временные (*lvalue*)?

Ответ (хором): ...

Решение: AddressOf

Достаточно реализовать конструктор принимающий аргумент по rvalue-ссылке. Тогда именно он будет выигрывать перегрузку в случае временных объектов.

```
template <class T>
const T* AddressOf(const T& value) {
    return &value;
}

template <class T>
const T* AddressOf(const T&&) = delete;

// ...

int x = 0;
auto array = new int[10];

AddressOf(x);           // Ok
AddressOf(array[5]);    // Ok
AddressOf(10);          // CE: use of deleted function
```

Решение: ссылочное поле класса

Достаточно реализовать функцию принимающую аргумент по rvalue-ссылке. Тогда именно она будет выигрывать перегрузку в случае временных объектов.

```
class A {  
    const int& cref_;  
  
public:  
    A(const int& value) : cref_(value) {}  
    A(const int&&) = delete;  
    const int& GetRef() { return cref_; }  
    const int* GetPtr() { return &cref_; }  
};  
  
int x = 0;  
auto array = new int[10];  
  
A a(x);           // ok  
A b(array[5]);    // ok  
A c(0);           // CE: use of deleted function
```

Мораль

По виду ссылки можно перегружать функции. Тогда результат lvalue выражений будет вызывать версию с левой ссылкой, а результат rvalue выражений - с правой

```
void f(int&);    // 1
void f(int&&);   // 2

int g();

int x = 0;

f(x);           // ???
f(0);           // ???
f(g());         // ???

f(++x);         // ???
f(x++);         // ???
```


Мораль

По виду ссылки можно перегружать функции. Тогда результат lvalue выражений будет вызывать версию с левой ссылкой, а результат rvalue выражений - с правой

```
void f(int&);    // 1
void f(int&&);   // 2

int g();

int x = 0;

f(x);           // 1
f(0);           // 2
f(g());         // 2

f(++x);         // 1
f(x++);         // 2
```

Конструктор перемещения и перемещающее присваивание

Напоминание

Вспомним конструктор копирования и копирующее присваивание стека

```
Stack<T>::Stack(const Stack<T>& other)
: buffer_(new T[other.capacity_])
, size_(other.size_) {

    for (size_t i = 0; i < size_; ++i) {    // O(N)
        buffer_[i] = other.buffer_[i];
    }
}

Stack<T>& Stack<T>::operator=(const Stack<T>& other) {
    if (this != &other) {
        // ...
        delete[] buffer_;
        // ... fill new buffer O(N)
    }
    return *this;
}
```

Проблема

```
Stack<int> stack;  
  
// ...  
  
auto lcopy = stack;           // копия stack: O(N)  
auto rcopy = Stack<int>(100); // до C++17: создание и копия объекта: O(N) + O(N)  
  
lcopy = stack;                // копия lcopy: O(N)  
rcopy = Stack<int>(100);      // создание и копия временного объекта: O(N) + O(N)
```

Создаем временный объект за $O(N)$, а затем его копируем за $O(N)$?!



Перемещение (C++11)

Идея: давайте напишем специальные версии конструктора и присваивания, которые принимают rvalue-ссылки (то есть работают со временными значениями)

```
Stack(Stack&& other) noexcept {  
    // ...  
}  
  
Stack& operator=(Stack&& other) noexcept {  
    // ...  
}  
  
auto rcopy = Stack(100); // Stack(Stack&&)  
rcopy = Stack(100);      // operator=(Stack&&)
```

А что они должны делать?

Перемещение (C++11)

```
Stack(Stack&& other) noexcept : buffer_(other.buffer_), size_(other.size_) {
    other.buffer_ = nullptr; // зачем?
    other.size_ = 0; // зачем?
}

Stack& operator=(Stack&& other) noexcept {
    if (this != &other) {
        delete[] buffer_;
        buffer_ = other.buffer_;
        size_ = other.size_;
        other.buffer_ = nullptr; // зачем?
        other.size_ = 0; // зачем?
    }
    return *this;
}

auto rcopy = Stack(100); // до C++17: создание и перемещение: O(N) + O(1)
rcopy = Stack(100);      //                создание и перемещение: O(N) + O(1)
```

Теперь лишних копирований не происходит!

Перемещение (C++11)

```
Stack(Stack&&) noexcept;  
Stack& operator=(Stack&&) noexcept;
```

Данные методы называются перемещающим конструктором и перемещающим присваиванием соответственно.

- Если вы не определили своего копирования, присваивания и деструктора (ничего из этого), то компилятор предоставит вам свой конструктор перемещения. Он вызовет конструктор перемещения для каждого из полей.
- Аналогично для перемещающего присваивания
- Для этих методов можно писать `= default`
- По причинам, которые обсудим позже, они должны быть `noexcept` !

Правило пяти (C++11)

- В современном C++ *правило трех* эволюционировало до *правила пяти*:

"Если класс требует реализации хотя бы одного метода из списка:

1. *Конструктор копирования*
2. *Конструктор перемещения*
3. *Копирующее присваивание*
4. *Перемещающее присваивание*
5. *Деструктор*

, то требуется реализовать их все"

- Как и "правило трех" это не правило языка, но для корректной работы программ следовать ему обязательно.

std::move (C++11)

Проблема

```
template <class T>
void Swap(T& x, T& y) {
    T tmp = x;
    x = y;
    y = tmp;
}
```

А что не так?

Проблема

```
template <class T>
void Swap(T& x, T& y) {
    T tmp = x;
    x = y;
    y = tmp;
}
```

```
Stack a(100'000'000, 0);
Stack b(100'000'000, 11);
```

```
Swap(a, b); // 3 копирования: ооочень долго
```

Как быть?

Идея

Идея проста: нужно не копировать, а перемещать

```
template <class T>
void Swap(T& x, T& y) {
    T tmp = x;    // переместить содержимое x в tmp
    x = y;        // переместить содержимое y в x
    y = tmp;      // переместить содержимое tmp в y
}
```

Но переменная это же всегда *lvalue*! Поэтому здесь всегда будет вызываться конструктор копирования.

Необходим способ заставить компилятор воспринимать выражение справа как *rvalue*.

`std::move` (C++11)

- В языке C++ есть преобразование из *lvalue* в *rvalue*.
- Это преобразование осуществляется с помощью функции `std::move`.
- `std::move` **не меняет** состояния объекта. Он лишь просит объект ненадолго *притвориться* временным (хотя таковым он являться не будет).



`std::move(x):`



`std::move` (C++11)

- Для результата `std::move` есть специальная категория - *xvalue* (частный случай *rvalue* - перемещаемый и с идентичностью)
- (Грубо) `std::move(x) <=> static_cast<type&&>(x)`

```
int x = 11;
std::move(x); // с x ничего не случится!

int& rx = std::move(x); // CE: справа rvalue!
int y = std::move(x); // Ok
int&& rrx = std::move(x); // Ok: справа rvalue

rrx = -1; // rrx связан со значением x!
std::cout << x << ' ' << rrx; // -1 -1
```

У базовых типов нет семантики перемещения, поэтому эти примеры скучные

`std::move` (C++11)

Поиграем со стеком

```
Stack x = y;    // конструктор копирования
y = x;          // копирующее присваивание

std::move(y);   // с y ничего не происходит!
Stack z = std::move(y); // конструктор перемещения
y = std::move(x); // перемещающее присваивание
```

В последнем случае не создано ни одного нового буфера!

Но `x` теперь пустой.

Пустым его сделал не `std::move`, а перемещающий оператор присваивания, реализованный нами!

Swap здорового человека

```
template <class T>
void Swap(T& x, T& y) {
    T tmp = std::move(x); // перемещаем содержимое x в tmp
    x = std::move(y);      // перемещаем содержимое tmp в x
    y = std::move(tmp);    // перемещаем содержимое tmp в y
}
```

```
Stack a(100'000'000, 0);
Stack b(100'000'000, 11);
```

```
Swap(a, b); // 3 перемещения: O(1)
```


Резюме

- У каждого выражения помимо типа есть категория значения (*lvalue* или *rvalue*)
- С *lvalue* могут связываться *lvalue*-ссылки, а с *rvalue* *rvalue*-ссылки
- Эти ссылки можно использовать для перегрузки функций, когда требуются разные действия со временными и не временными объектами
- Основное применение - перемещающие конструкторы и присваивания, с помощью которых реализуется *семантика перемещения*
- Чтобы заставить компилятор вызвать *rvalue* перегрузку для *lvalue* выражений, нужно воспользоваться `std::move`