**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет прикладной математики и информатики**

**Кафедра информационных систем управления**

### 

Кипер Владислав Александрович

**РЕФЕРАТ**

**на тему** “**Перемещающий конструктор и семантика перемещения**”

студента 1 курса 2 группы

**Преподаватель**

Баранов Геннадий Аркадьевич

Минск, 2017

**СОДЕРЖАНИЕ**

Введение………………………………………………………………………2

Свойства выражений….………….…………………………………………3

Rvalue ссылки...………………………………………………………………7

Семантика перемещения. Перемещающий конструктор…………..…9

std::move……….…………………………………………………………….16

Заключение………………………………………………………………….18

Ссылки………………………………………………………………………..19

**ВВЕДЕНИЕ**

В стандарте С++11 введены [rvalue ссылки](https://pro-prof.com/forums/topic/cplusplus_references), за счет которых возможно реализовать отдельные функции для обработки обычных (живущих дольше выражения) и временных объектов. Такой подход позволяет написать более оптимальный код, так как, зная, что аргументом является временный объект, который все равно будет уничтожен после обработки, мы можем использовать данные этого объекта, вместо создания новых. Такое поведение называется семантикой перемещения, для его реализации служат перемещающие конструкторы, принимающие на вход *rvalue* ссылку.

**СВОЙСТВА ВЫРАЖЕНИЙ**

Прежде чем говорить о новшествах, необходимо раскрыть тему свойств выражений. Итак, начнем, - lvalue и rvalue существуют достаточно давно, но не каждый С++ 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 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 прошло много времени, 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, так и rvalue выражение, а следовательно, нельзя говорить, что свойство выражения определяется его пространственным расположением.

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. Есть вероятность, что адрес все-таки можно получить, потому что некоторые компиляторы дают это сделать, однако это является нарушением стандарта C++. Вообще говоря, этот адрес не будет иметь никакого смысла, т.к. 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:**

#include <iostream>  
using namespace std;  
int& foo()  
{  
 int x = 5;  
 return x;  
}  
int bar()  
{  
 return 1;  
}  
void main()  
{  
 int a = 0, b = 0;  
*/\* Мы не можем получить адрес (a+b), ибо результатом сложения будет временный объект, доступ к которому не может быть получен за пределами строки выполнения оканчивающейся ';' \*/*  
 &(a + b); */\* Ошибка C2102 : для "&" требуется левостороннее значение \*/*  
 &a += b; */\* Без ошибок, потому что мы можем получить адрес результата этого выражения. \*/*  
 &foo(); *// Без ошибок*  
 &bar(); */\* Ошибка C2102 : для "&" требуется левостороннее значение \*/*  
 &++a; *// Без ошибок*  
 &a++; */\*Ошибка C2102 : для "&" требуется левостороннее значение \*/*  
 return;  
}

Листинг программы 1.

Хотя приведенное выше методика распознавания rvalue не универсальна, она может позволить выработать рефлекс на rvalue. Однако, необходимо отметить, что считать временный объект и rvalue синонимами - неправильно, ибо rvalue\lvalue - это свойство выражения, а не объекта.

**Например:**

bar();

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

const int& lvalue = bar();

Объект возвращаемый bar() все еще является временным, но, согласно стандартам С++, 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.

В итоге можно утверждать, что rvalue-выражения - это:

* Выражения, обозначающие временные объекты. В частности, результат вызова функций, возвращающих объекты не по ссылке; результат встроенных операций “+”, “-”. “\*”, “/” и т.д.; явное создание временной переменной; преобразования не к ссылочным типам и т.д.
* Результат встроенной операции взятия адреса (&) — rvalue типа указатель.
* Результат встроенных постфиксных операций ++, --.
* Литералы за исключением строковых.
* Константы перечислений.

Далее рассмотрим то, что предлагает новый стандарт, а именно: следующее разделение

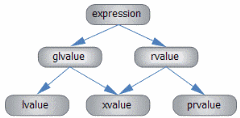


Рисунок 1 - схема разделения выражений

Новый стандарт вводит новые понятия и расширяет идею rvalue\lvalue. Кратко рассмотрим каждый по очереди:

**lvalue** - здесь никаких изменений, lvalue свойство обозначает то же самое, что обозначает в C++03.

**xvalue** - 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 rvalue ссылками, которые будут рассмотрены далее.

**prvalue** - в новом стандарте так обозвали rvalue из С++03.

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

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

Отметим еще раз, что xvalue появляется лишь тогда, когда в результате выражения фигурируют *rvalue ссылки.*

**RVALUE ССЫЛКИ**

Одним из самых значительных изменений в ядре языка по праву можно считать введение rvalue ссылок.

Rvalue ссылка – это составной тип, очень похожий на традиционную ссылку в C++. Они получили свое имя по аналогии с давно знакомыми любому С++ программисту ссылками. Теперь эти, "старые" ссылки, именуются не иначе как lvalue ссылки, т.е. в дальнейшем, чтобы различать эти два типа, 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.

Отличие lvalue от rvalue ссылок состоит, во-первых, в способе записи: если lvalue ссылки используют лексему "&" для своей декларации, то rvalue использует двойной амперсанд "&&" (важно отметить, что данное обозначение не является ссылкой на ссылку). Во-вторых, они отличаются по типам объектов, на которые они могут ссылаться:

Type& // может ссылаться на любое не константное lvalue.  
 const Type& // может ссылаться на любое выражение.  
 Type&& // может ссылаться на не константные xvalue и prvalue.  
 const Type&& // может ссылаться на любое выражение, кроме lvalue.

Здесь можно рассмотреть довольно много примеров, однако далее приведено лишь несколько из них:

int&& xvalue\_func()  
{  
 return 5;  
}  
   
int& lvalue\_func()  
{  
 static int i = 0;  
 return i;  
}  
   
int prvalue\_func()  
{  
 return 5;  
}  
int main()  
{  
 double d = 0.0;  
 const int i = 0;  
 //#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 не может быть привязана к переменной,  
 чей интегральный тип не совпадает с типом ссылки \*/*  
 int& type\_mismatch\_lvalue = d;  
 */\* #5:OK, rvalue ссылка может быть привязана к переменной, чей  
 интегральный тип не совпадает с типом ссылки \*/*  
 int&& type\_mismatch\_rvalue = d;  
 //#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&& rvalue4 = 0;  
 //#9:OK, const Type& может быть привязано к любому выражению  
 const int& const\_lvalue2 = prvalue\_func();  
 //#10:Error, const Type&& не может быть привязано к lvalue  
 const int&& const\_rvalue1 = lvalue\_func();  
}

Листинг программы 2.

Следует обратить внимание на пример #4: в этом примере 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. После конвертации получается временный объект типа int, и в результате выражение из lvalue превращается в prvalue. Но 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.

**СЕМАНТИКА ПЕРЕМЕЩЕНИЯ**.

**ПЕРЕМЕЩАЮЩИЙ КОНСТРУКТОР**

Наконец, перейдем к практическому применению изложенного выше материала - семантике перемещения. Это одно из самых важных и нужных новшеств, которые привнесли rvalue ссылки в C++.

Ссылки rvalue поддерживают реализацию *семантики перемещения*, которая 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 ссылаться из других мест в программе.

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 реализовать семантику перемещения, в классе обычно следует задать *конструктор перемещения* и, при необходимости, оператор присваивания перемещения (operator=). 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.

Рассмотрим на примере, как создать *конструктор перемещения* и оператор присваивания перемещения для класса. Пример будет построен на основе приведенного ниже класса C++ DATA, который управляет буфером памяти.

**#include <iostream>**

**#include <vector>  
using namespace std;  
class DATA  
{  
private:  
 int amount; // Объем данных  
 int\* data; // Данные  
public:   
 // Конструктор   
 DATA(int amount): amount(amount), data(new int[amount])  
 {  
 cout << "Конструктор. Создаем данные объема "  
 << amount<< "." << endl;  
 {  
 // Деструктор.   
 ~DATA()  
 {  
 cout << "Деструктор. ";  
 if (data != nullptr)  
 {  
 cout << " Удаляем данные объема "<<amount<<"."<<endl;  
 delete[] data;  
 }  
 }  
 // Конструктор копирования  
 DATA(const DATA& other): amount(other.amount), data(new int[other.amount])  
 {  
 cout << "Конструктор копирования. Копируем данные объема "  
 << other.amount<< endl;  
 copy(other.data, other.data + amount, data);   
 }  
 // Оператор присваивания   
 DATA& operator=(const DATA& other)  
 {  
 cout << "Оператор присваивания." << endl;  
 if (this != &other)  
 {  
 delete[] data;  
   
 amount= other.amount;  
 data = new int[amount];  
 copy(other.data, other.data + amount, data);  
 }  
 return \*this;  
 }   
};** Листинг программы 3

В следующих процедурах описывается создание конструктора перемещения и оператора присваивания перемещения для этого примера класса C++.

Создание конструктора перемещения для класса C++

1. Определите пустой метод конструктора, принимающий в качестве параметра ссылку rvalue на тип класса.
2. В конструкторе перемещения присвойте создаваемому объекту данные-члены класса из исходного объекта.
3. Присвойте данным-членам исходного объекта значения по умолчанию. Это не позволяет деструктору многократно освобождать ресурсы (например, память).

### Создание оператора присваивания перемещения для класса C++

1. Определите пустой оператор присваивания, принимающий в качестве параметра ссылку rvalue на тип класса и возвращающий ссылку на тип класса.
2. В операторе присваивания перемещения добавьте условный оператор, который не выполняет никакой операции при попытке присвоить объект самому себе.
3. В условном операторе освободите все ресурсы (такие как память) из объекта, которому производится присваивание.
4. Выполните шаги 2 и 3 из первой процедуры, чтобы переместить данные-члены из исходного объекта в создаваемый объект:
5. Верните ссылку на текущий объект, как показано в следующем примере:

Итого имеем:

// Перемещающий конструктор  
DATA(DATA&& other) :data(nullptr), amount(0)  
 {  
 cout << "Перемещающий конструктор. Переместили данные объема " << other.amount << "." <<endl;  
 // Копируем указатель и переменную объема данных   
 data = other.data;  
 amount = other.amount;  
 // Освобождаем указатель объекта, деструктору придется   
 // освобождать память много раз   
 other.data = nullptr;  
 other.amount = 0;  
 }

Листинг программы 4.

// Перемещающий оператор присваивания

DATA& operator=(DATA&& other)  
 {  
 cout << "Оператор присваивания. Перемещаем данные объема "  
 << other.amount << "." << endl;  
 if (this != &other)  
 {  
 delete[] data;  
 data = other.data;  
 amount = other.amount;  
 other.data = nullptr;  
 other.amount = 0;  
 }  
 return \*this;

Листинг программы 5.

Учитывая листинги программы 3, 4 и 5, в следующем примере покажем, как семантика перемещения может повысить производительность приложений. В примере добавляются два элемента в объект-вектор, а затем вставляется новый элемент между двумя существующими элементами.

**void main()  
{  
 vector<DATA> v;  
 v.push\_back(DATA(30));  
 v.push\_back(DATA(40));**

**v.insert(v.begin() + 1, DATA(50));  
 return;  
}**

В этом примере выводятся следующие данные:

Конструктор. Создаем данные объема 30.

Перемещающий конструктор. Переместили данные объема 30.

Деструктор.

Конструктор. Создаем данные объема 40.

Перемещающий конструктор. Переместили данные объема 30.

Деструктор.

Перемещающий конструктор. Переместили данные объема 40.

Деструктор.

Конструктор. Создаем данные объема 50.

Перемещающий конструктор. Переместили данные объема 30.

Перемещающий конструктор. Переместили данные объема 40.

Деструктор.

Деструктор.

Перемещающий конструктор. Переместили данные объема 50.

Перемещающий конструктор. Переместили данные объема 40.

Оператор присваивания. Перемещаем данные объема 50.

Оператор присваивания. Перемещаем данные объема 40.

Деструктор.

Деструктор.

Деструктор.

Удаляем данные объема 30.

Деструктор.

Удаляем данные объема 50.

Деструктор.

Удаляем данные объема 40.

Необходимо отметить, что если бы мы использовали семантику перемещения, а именно перемещающий конструктор, то в данном примере вывод был бы следующий:

Конструктор. Создаем данные объема 30.

Конструктор копирования. Копируем данные объема 30

Деструктор.

Удаляем данные объема 30.

Конструктор. Создаем данные объема 40.

Конструктор копирования. Копируем данные объема 30

Деструктор.

Удаляем данные объема 30.

Конструктор копирования. Копируем данные объема 40

Деструктор.

Удаляем данные объема 40.

Конструктор. Создаем данные объема 50.

Конструктор копирования. Копируем данные объема 30

Конструктор копирования. Копируем данные объема 40

Деструктор.

Удаляем данные объема 30.

Деструктор.

Удаляем данные объема 40.

Конструктор копирования. Копируем данные объема 50

Конструктор копирования. Копируем данные объема 40

Оператор присваивания.

Оператор присваивания.

Деструктор.

Удаляем данные объема 40.

Деструктор.

Удаляем данные объема 50.

Деструктор.

Удаляем данные объема 30.

Деструктор.

Удаляем данные объема 50.

Деструктор.

Удаляем данные объема 40.

Версия этого примера, в которой используется семантика перемещения, более эффективна, чем версия, в которой эта семантика не используется, поскольку в ней выполняется меньше операций копирования, выделения памяти и освобождения памяти.

Было показано, что перемещающий конструктор является облегченной версией конструктора копирования, однако иногда реализуют перемещающий конструктор для классов, копирование в которых запрещено. Ярчайший пример — std::unique\_ptr<>, представляющий собой один из умных указателей стандартной библиотеки, реализующий [идиому RAII](https://pro-prof.com/forums/topic/resources-control-objects-raii). Суть этого класса заключается в том, что он владеет единственным указателем на некоторый объект (и автоматически разрушает объект в определенных ситуациях). Указатель должен быть единственным, следовательно конструктор копирования не должен быть доступен, однако перемещение для этого класса вполне логично (мы можем передавать владение указателем от одного объекта к другому — поэтому unique\_ptr, в частности, может быть использован в качестве возвращаемого значения функции. Другими примерами такого поведения из стандартной библиотеки являются классы std::fstream и std::thread — копирование для них лишено смысла, однако передача владения файлом из одной функции в другую может быть логична.

Таким образом, семантика перемещения является не только средством повышения эффективности программ, но и позволяет реализовать передачу владения объектом в случаях, когда копирование запрещено (является очень длительной операцией или вообще лишено смысла).

Стоит отметить, что не для всех классов имеет смысл писать перемещающий конструктор. Так, вы не сможете, например, повысить эффективность за счет перемещения для класса, не владеющего внешним ресурсом. Ведь повышение эффективности обеспечивается заменой копирования этого ресурса перемещением.

**STD::MOVE**

В ряде случаев нужно явно указать компилятору, что объект является временным — сделать это можно с помощью функции std::move, выполняющей приведение типа к rvalue (эквивалентной static\_cast<T&&>). Если вам нужно переместить именованный (не временный) объект — используйте std::move, т.к. в противном случае будет вызван конструктор копирования.

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:

void myswap(vector<T>v1, vector<T>v2)  
{  
 vector<T> v3;  
 v3 = v1; // теперь у нас две копии вектора v1  
 v1 = v2; // теперь у нас две копии вектора v2  
 v2 = v3; // теперь у нас две копии вектора v3, т.е. вектора v1  
}

Листинг пограммы 6.

В действительности нам не нужны копии v1 или v2, мы просто хотели обменять их. Давайте попробуем еще раз:

void myswap(vector<T>v1, vector<T>v2)  
{  
 vector<T> v3;  
 v3 = move(v1);   
 v1 = move(v2);  
 v2 = move(v3);   
}

Листинг пограммы 7.

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).

Функция move в действительности выполняет весьма скромную работу. Еe задача состоит в том, чтобы принять либо lvalue, либо rvalue параметр, и вернуть его как rvalue без вызова конструктора копирования, т.е. std::move не перемещает объект, а лишь выполняет приведение типа, которое позволяет вызвать перемещающий конструктор.

**ЗАКЛЮЧЕНИЕ**

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 операторов присваивания и конструктора перемещения связаны с некоторыми сложностями(написание нового кода, использование этого кода), move семантика уже используется в stl и, возможно, будет использоваться во всех известных библиотеках, которыми занимаются сознательные разработчики, тем более, что rvalue ссылки уже сейчас поддерживаются главными С++ компиляторами (gcc и MSVC).

Move-конструкторы и операторы присваивания вещь, безусловно, полезная в плане оптимизации производительности. Во многих контейнерах STL происходит копирование временных объектов, например, при реаллокации в связи с изменением объема контейнера. Move семантика позволяет добиться существенного прироста производительности. Если в конструкторе копирования происходит копирование каких-то массивов данных, которые представлены указателем, то это явный признак того что надо реализовать еще move-конструктор.

**ССЫЛКИ**