Сематника переноса (move semantics)

М. А. Ложников

15 апреля 2019 г.

Это предварительная невыверенная версия. Читайте на свой страх и риск.

1 Типы выражений и ссылок

Про понятия, обсуждаемые в этом разделе можно прочитать в статье [1].

1.1 Понятия lvalue и rvalue

В языке C++ есть два типа выражений: *lvalue* и *rvalue*. Подробное их определение доступно на cppreference [2]. На практике достаточно знать следующее:

- Если можно взять адрес выражения (`&`) то это выражение есть lvalue.
- Если тип выражения является lvalue ссылкой (то есть обычной ссылкой вида `T&` или `const T&`), то такое выражение есть lvalue.
- Остальные выражения являются rvalue. Обычно rvalue выражения соответствуют временным объектам, например, объектам, которые возвращает функция, или объектам, созданным посредством операций неявного преобразования типа. Большинство литералов (например, `10` или `5.3`) также являются rvalue выражениями.

```
int Negative(int number) {
  return -number;
}

/* x --- lvalue выражение (можно взять адрес), 5 --- rvalue выражение (литерал),
  x + y --- rvalue выражение (временный объект), Negative(z) --- тоже rvalue выражение
  (временный объект). */
int x = 5;
int y = 10;
int z = x + y;
z = Negative(z);
```

1.2 Rvalue и lvalue ссылки

Обычная ссылка в C++ вида `T&` или `const T&` называется *lvalue ссылкой*. В стандарте C++11 появились также *rvalue ссылки*, которые имеют вид `T&&` (два амперсанда пишутся слитно), где `T` — любой нессылочный тип данных. Точно также как и обычная lvalue ссылка, rvalue ссылку необходимо инициализировать при объявлении. Причём, rvalue ссылку можно инициализировать только rvalue выражением, а lvalue ссылку lvalue выражением и при некоторых ограничениях (если это ссылка на константу, то есть имеет вид `const T&`) rvalue выражением. Рассмотрим пример.

```
struct Person {
  std::string name;
  std::string lastName;
  int age;
};
Person MakePerson() { // Функция возвращает временный объект
  return {"Ivan", "Petrov", 25};
}

Person person = {"Petr", "Ivanov", 52};
/* Rvalue ссылки можно инициализировать rvalue выражениями (в данном случае временным
   объектом). */
Person&& rValueRef = MakePerson();

// Lvalue ссылку на константу также можно инициализировать rvalue выражением.
const Person& constLValueRef = MakePerson();

/* Ошибка компиляции, именованная переменная (person) является lvalue выражением. */
// Person&& rValueRef2 = person;

/* lvalue ссылка инициализируется lvalue выражением. */
Person& lValueRef = person;

/* Ошибка компиляции. Нельзя инициализировать обычную lvalue ссылку rvalue
  выражением. */
// Person& lValueRef = MakePerson();
```

Отметим, что тип данных выражения (тип ссылки) не говорит о том, является выражение lvalue или rvalue. В частности rvalue ссылка может быть как rvalue выражением, так и lvalue выражением. Рассмотрим пример.

```
/* Функция MakePerson() возвращает временный объект. Переменная rValueRef является
  rvalue ссылкой, однако само выражение rValueRef является lvalue выражением,
  поскольку от него можно взять адрес. */
Person&& rValueRef = MakePerson();

/* Всё впорядке, rvalue ссылка rValueRef типа Person&& сама по себе
  является lvalue выражением. */
Person& lValueRef = rValueRef;

/* Ошибка компиляции. Rvalue ссылку нельзя инициализировать lvalue выражением. */
// Person&& rValueRef2 = rValueRef;

/* Всё в порядке. Приведение типа выдаёт rvalue выражение. */
Person&& rValueRef3 = static_cast<Person&&>(rValueRef);
```

Способ получения rvalue выражения (например, для инициализации rvalue ссылки) посредством `static_cast` не очень хорош, он не всегда работает, например, в случае, если бы тип `Person` был шаблонным параметром. Правильный способ — использовать функцию `std::move()` из заголовочного файла `<utility>`.

```
Person&& rValueRef = MakePerson();
Person& lValueRef = rValueRef;

/* Правильный способ. */
Person&& rValueRef2 = std::move(rValueRef);

/* Из lvalue ссылки тоже можно получить rvalue ссылку. */
Person&& rValueRef3 = std::move(lValueRef);
```

1.3 Универсальные ссылки

Подробнее про эту тему можно прочитать в статье [3] (оригинал на английском [4]). Универсальные ссылки легко перепутать с rvalue ссылками, поскольку они объявляются похожим образом.

```
Person&& rValueRef = MakePerson(); // rValueRef --- rvalue ссылка
auto&& universalRef = rValueRef;   // Не rvalue ссылка

// Тип аргумента функции выводится не полностью.
template<typename T>
void DoSomething(std::vector<T>&& rValueRef2); // Это rvalue ссылка.

// Тип аргумента функции выводится полностью.
template<typename T>
void DoSomething(T&& universalRef2); // Это не rvalue ссылка.
```

Таким образом, универсальными ссылками назовём либо переменные, объявленные как `auto&&` либо аргументы функций вида `T&&`, где тип данных `T` полностью выводится при вызове функции. Такие ссылки могут вести себя либо как lvalue ссылки, либо как rvalue ссылки в зависимости от того, чем их инициализировали. Правила довольно просты:

- Если универсальную ссылку инициализировали rvalue выражением, то она становится rvalue ссылкой.
- Если универсальную ссылку инициализировали lvalue выражением, то она становится lvalue ссылкой.

Рассмотрим пример.

```
Person person = MakePerson();

/* Инициализируется rvalue выражением, значит является rvalue ссылкой. Переменная
  rValueRef имеет тип данных Person&&. */
auto&& rValueRef = MakePerson();

/* Инициализируется lvalue выражением, значит является lvalue ссылкой. Переменная
  lValueRef имеет тип Person&. */
auto&& lValueRef = person;

/* Сама по себе rvalue ссылка rValueRef является lvalue выражением (от неё можно взять
```

```
  адрес), значит lValueRef2 является lvalue ссылкой и имеет тип данных Person&. */
auto&& lValueRef2 = rValueRef;
```

Предположим, функция `DoSomething()` принимает ссылку и передаёт её в другую функцию `DoSomethingHelper()`, также принимающую универсальную ссылку. Как правильно передать параметр? Хотелось бы сделать так, что в случае lvalue ссылки передавалась также lvalue ссылка, а в случае rvalue ссылки передавалась rvalue ссылка. Сложность заключается в том, что сама по себе универсальная ссылка является lvalue выражением, а если сделать от неё `std::move()`, то всегда будет получено rvalue выражение. В таких случаях следует использовать функцию `std::forward()` из заголовочного файла `<utility>`.

```
template<typename T>
void DoSomethingHelper(T&& helperParam);

template<typename T>
void DoSomething(T&& param) {

  /* Ссылка param есть lvalue выражение, поэтому тип данных helperParam будет
    lvalue ссылкой типа T&. */
  DoSomethingHelper(param);

  /* std::move() возвращает rvalue выражение, поэтому тип данных helperParam будет
    rvalue ссылкой типа T&&. */
  DoSomethingHelper(std::move(param));

  /* std::forward() возвращает lvalue выражение, если param является lvalue ссылкой
    и rvalue выражение, если param является rvalue ссылкой. */
  DoSomethingHelper(std::forward(param));
}
```

2 Семантика переноса

Рассмотрим примеры использования rvalue ссылок. Одним из наиболее важным нововведением стандарта C++11 стала возможность перемещения объекта класса (например, внутрь какого-либо контейнера) без копирования.

2.1 Порядок вызова операций перемещения

Предположим, что есть некоторый объект, который довольно долго копировать. Требуется переместить его в контейнер. При этом операция перемещения должна происходить быстро, она не должна создавать копию объекта.

В C++11 для этого появился стандартный механизм. Для того, чтобы он работал необходимо, чтобы у перемещающего объекта был определён специальный *перемещающий конструктор*, а также перемещающая операция присваивания. Перемещающий конструктор и перемещающий оператор присваивания принимают rvalue ссылку на объект. Они должны быть написаны специальным образом так, чтобы перемещать “внутренности” старого объекта в новый. Рассмотрим пример.

```
class Logger {
 public:
```

```
  Logger(const std::string& info) :
    info(info)
  { }

  Logger(const Logger& other) :                // copy-конструктор
    info(other.info)
  {
    std::cout << "Logger '" << info << "': copy-constructor!" << std::endl;
  }

  Logger(Logger&& other) :                     // перемещающий конструктор
    info(std::move(other.info))
  {
    /* В списке инициализации был вызван перемещающий конструктор класса std::string. */
    std::cout << "Logger '" << info << "': move-constructor!" << std::endl;
  }

  Logger& operator=(const Logger& other) {     // операция копирующего присваивания
    info = other.info;

    std::cout << "Logger '" << info << "': copy assignment operator!" << std::endl;

    return *this;
  }

  Logger& operator=(Logger&& other) {          // операция перемещающего присваивания
    /* Вызываем операцию перемещающего присваивания для класса std::string. */
    info = std::move(other.info);

    std::cout << "Logger '" << info << "': move assignment operator!" << std::endl;

    return *this;
  }
 private:
  std::string info;
};

int main() {
  Logger mainLogger("MainLogger");
  Logger secondLogger("SecondLogger");

  /* Copy-конструктор. loggerCopy точная копия mainLogger. */
  Logger loggerCopy(mainLogger);

  /* Функция std::move() возвращает rvalue выражение, поэтому вызывается конструктор
     перемещения, принимающий rvalue ссылку. Однако, если у класса нет перемещающего
     конструктора, то будет вызываться обычный конструктор копирования поскольку
     ссылку на константу (которую принимает copy-конструктор) можно также
     инициализировать rvalue выражением. */
  Logger newLogger(std::move(mainLogger));

```

```
54   /* Сама функция std::move() ничего никуда не перемещает. Она просто способствует
55     вызову перемещающего метода, принимающего rvalue ссылку. В данном случае никакие
56     методы класса Logger не вызываются. Никакого перемещения не произойдёт. */
57   std::move(newLogger);
58
59   /* Операция копирующего присваивания. */
60   loggerCopy = newLogger;
61
62   /* Операция перемещающего присваивания. Если её нет, то вызовется обычная операция
63     копирующего присваивания поскольку lvalue ссылка на константу также может быть
64     инициализирована rvalue выражением. */
65   loggerCopy = std::move(secondLogger);
66   return 0;
67 }
```

Отметим, что компилятор автоматически создаёт конструктор перемещения и операцию копирующего перемещения. Однако, он это делает только в том случае, если не был определён пользовательский copy-конструктор или пользовательская операция копирующего присваивания. В этом случае компилятор просто применит перемещающий конструктор или перемещающую операцию присваивания поэлементно. Для классов из стандартной библиотеки написаны правильные перемещающие операции. Таким образом, если поля класса являются стандартными типами STL (например, контейнерами), то перемещающие операции обычно ненужно переопределять. Однако, если класс содержит указатели, операции перемещения необходимо переопределить.

2.2 Написание правильных операций перемещения

Итак, перемещающие операции должны перемещать внутренности объекта из старого в новый. Рассмотрим пример таких операций на базе шаблона `Vector`.

```
1  template<typename T>
2  Vector {
3   public:
4    Vector() :  // Стандартный конструктор
5      data(nullptr),
6      size(0),
7      capacity(0)
8    { }
9
10   explicit Vector(std::size_t size) : // Специальный конструктор
11     data(new T[size]),
12     size(size),
13     capacity(size)
14   {
15     for (std::size_t i = 0; i < size; i++)
16       data[i] = T(); // инициализация значением по-умолчанию.
17   }
18
19   ~Vector() { // Деструктор
20     if (data)
21       delete[] data;
```

```
22  }
23
24  Vector(const Vector& other) { // Copy-конструктор
25    // Реализация конструктора копирования (опустим для краткости).
26  }
27
28  Vector& operator=(const Vector& other) {
29    // Реализация операции копирующего присваивания
30  }
31
32  /* Перемещающий конструктор */
33  Vector(Vector&& other) :
34    data(other.data),
35    size(other.size),
36    capacity(other.capacity)
37  {
38    /* Мы просто скопировали указатель, теперь следует очистить объект other,
39      чтобы не произошло двойного освобождения памяти в деструкторе. Таким образом
40      и произойдёт перемещение данных. */
41    other.data = nullptr;
42    other.size = 0;
43    other.capacity = 0;
44
45    return *this;
46  }
47
48  Vector& operator=(Vector&& other) {
49    /* Здесь немного сложнее. Это не конструктор, в поле data могут содержаться
50      данные. Можно просто поменять его местами с полем other.data (с полем size
51      сделать то же самое). А можно освободить память в data, если там что-то было.
52      Реализуем второй способ. */
53    if (data)
54      delete[] data;
55
56    // Теперь можно перемещать данные. */
57    data = other.data;
58    size = other.size;
59    capacity = other.capacity;
60
61    other.data = nullptr;
62    other.size = 0;
63    other.capacity = 0;
64
65    return *this;
66  }
67 private:
68  T* data;
69  std::size_t size;
70  std::size_t capacity;
71
72  /* Остальные поля и методы... */
```

```
};
```

2.3 Перемещение объекта внутрь контейнера

Теперь предположим, что перемещаемый объект поддерживает перемещающие операции. Напишем метод, который может запихнуть его в вектор без копирования.

```
template<typename T>
class Vector {
  /* Конструкторы и прочее... */

 public:
  /* Параметр item является rvalue ссылкой, а не универсальной ссылкой поскольку тип T
     выводится при подстановке в шаблон класса, а не в шаблон функции PushBack(). */
  void PushBack(T&& item) {
    if (!data) {
      data = new T[1];
      capacity = 1;
    }

    if (size >= capacity) { // Выделим память, если не хватает.
      T* tmp = new T[capacity * 2];

      for (std::size_t i = 0; i < size; i++) {
        tmp[i] = std::move(data[i]); // Перемещаем данные из старого массива в новый.
      }

      delete[] data;
      data = tmp;
      capacity *= 2;
    }

    /* Вызываем операцию перемещающего присваивания для объекта. Rvalue ссылка item
       является lvalue выражением, поэтому нужно опять вызвать std::move(), чтобы
       получить rvalue выражение. */
    data[size] = std::move(item);
    size++;
  }
};
```

Контейнеры стандартной библиотеки поддерживают перемещения в них. Рассмотрим пример.

```
#include <iostream>
#include <vector>
#include <string>

int main() {
  std::vector<std::string> vec;
  std::string line;
```

```
  int N = 0;

  std::cin >> N;

  for (int i = 0; i < N; i++) {
    if (!std::getline(std::cin, line)) {
      std::cerr << "Can't read line!" << std::endl;
      return 0;
    }

    std::cout << "New line: '" << line << "'" << std::endl;
    vec.push_back(std::move(line));
    std::cout << "Line is empty now: '" << line << "'" << std::endl;
  }

  std::cout << "Contents of vector:" << std::endl;
  for (std::string& item : vec) {
    std::cout << "Line: '" << item << "'" << std::endl;
  }
  return 0;
}
```

2.4 Операции по-умолчанию

Пусть класс переопределяет конструктор копирования и операцию копирующего присваивания. Кроме того, предположим, что для класса подходят стандартные операции перемещения. Однако, как было отмечено выше, в этом случае, компилятор не будет их создавать. В C++ есть способ заставить его создать стандартные перемещающие операции.

```
class SomeClass {
 public:
  SomeClass(const SomeClass& other) {
    // Переопределение copy-конструктора
  }

  SomeClass& operator=(const SomeClass& other) {
    // Переопределение операции копирующего присваивания
  }

  /* = default заставляет создать конструктор перемещения по-умолчанию. */
  SomeClass(SomeClass&&) = default;

  /* Аналогично создаём операцию перемещающего присваивания по-умолчанию. */
  SomeClass& operator=(SomeClass&&) = default;
};
```

2.5 Некопируемый тип

Для некоторых типов, например, для оболочек над файловыми дескрипторами не имеет смысла операция копирования. В таком случае её можно запретить, тогда компилятор будет выдавать ошибки

при попытке её использования.

```
class Uncopyable {
 public:
  /* = delete говорит компилятору о том, что не нужно создавать реализацию по-умолчанию.
     Таким образом, соответствующий метод будет удалён из класса. */
  Uncopyable(const Uncopyable&) = delete;
  Uncopyable& operator=(const Uncopyable&) = delete;
};
```

3 Различные оптимизации

В этом разделе рассмотрим различные оптимизации, которые производит компилятор для того, чтобы избежать ненужного копирования.

3.1 Copy-elision

Предположим, что некоторая функция возвращает временный объект, который затем сохраняется в переменную. Что в этом случае произойдёт? Будут ли вызваны операции копирования или перемещения? Рассмотрим пример с использованием ранее написанного класса `Logger`.

```
Logger MakeLogger(const std::string& name) {
  return Logger(name); // Возвращаем временный объект
}

int main() {
  Logger someLogger = MakeLogger("SomeLogger");
  return 0;
}
```

Оказывается, что в данном случае не происходит ни операции копирования, ни операции перемещения. Объект, который возвращает функция `MakeLogger()`, просто продолжает жить под именем `someLogger`. Это свойство называется *copy elision*.

3.2 Named return value optimization

Теперь рассмотрим аналогичный пример, когда функция возвращает локальную переменную.

```
Logger MakeLogger(const std::string& name) {
   Logger logger(name); // logger - локальная переменная

  return logger;
}

int main() {
  Logger someLogger = MakeLogger("SomeLogger");
  return 0;
}
```

В данном случае также не происходит ни операции копирования ни операции перемещения. Переменная `logger`, объявленная в функции `MakeLogger()`, продолжает жить под именем `someLogger`. Это свойство называется *named return value optimization*.

Отметим, что данная оптимизация работает только для локальных переменных. Приведём пример, когда она не работает.

```
Logger MakeLogger(const std::string& name) {
  /* Здесь сработает конструктор перемещения при создании пары. Созданный Logger
    будет перемещён в поле first пары p, но это не ошибка. */
  std::pair<Logger, int> p(Logger(name), 0);

  /* Ошибка заключается в том, что p.first не является локальной переменной, это поле
    пары p. Таким образом, для него не работает named return value optimization. При
    возврате значения из функции произойдёт копирование. */
  return p.first;
}

int main() {
  /* Здесь произойдёт копирование. */
  Logger someLogger = MakeLogger("SomeLogger");
  return 0;
}
```

Список литературы

[1] http://thbecker.net/articles/rvalue_references/section_01.html

[2] https://en.cppreference.com/w/cpp/language/value_category

[3] https://habr.com/ru/post/157961/

[4] https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers

[5] https://en.cppreference.com/w/cpp/language/reference

[6] https://en.cppreference.com/w/cpp/language/move_constructor

[7] https://en.cppreference.com/w/cpp/language/move_assignment