
Яндекс

Яндекс
карты

Rvalue references in C++11

Дмитрий Прокопцев
Яндекс.Карты, старший разработчик

Кто мы такие

Пишем на C++11 с 2010 года;

Продирались через черновики стандарта;

Набили шишек и получили опыт :)

Краткое содержание

- Зачем всё это нужно?
- Новые ссылки в языке
- Перемещение классов
- Как быть с исключениями
- Универсальные ссылки

Зачем?

- **Скорость работы:**
  - std::vector<X> v = vector_with_400M_items();
- **Семантическая ясность:**
  - std::unique_ptr<X> make_x();
  - std::thread spawn_thread();
  - std::iostream tcp_connect(const std::string& addr);

Но ведь уже было?

```
template<class T> class auto_ptr {
private:
    T* held;

public:
    // needed for ‘auto_ptr<X> y2 = y1’
    auto_ptr(auto_ptr<X>& other) {
        held = other.held;
        other.held = 0;
    }

    // see next page...
};
```

Но ведь уже было?

```
template<class T> class auto_ptr { // cont'd
public:

    // needed for 'auto_ptr<X> x = f()'
    struct auto_ptr_ref {
        auto_ptr<T>* ptr;
        auto_ptr_ref(auto_ptr<T>* self): ptr(self) {}
    };

    operator auto_ptr_ref() { return auto_ptr_ref(this); }

    auto_ptr(auto_ptr_ref ref) {
        held = ref.ptr->held;
        ref.ptr->held = 0;
    }
};
```

Но ведь уже было?

- **Одни и те же 10 строк для каждого класса;**
- **Семантическая неоднозначность:**
  - std::vector<char> v2 = v1; // копирование
  - std::auto_ptr<X> p2 = p1; // перемещение
  - template<class X> void f(X& x) {
    X y = x; // копирование? перемещение?
    }

Ревизия ссылок

- **T::T(const T&)**
  - Копирование;
  - Может быть вызвано и от lvalues, и от временных объектов;

- **T::T(T&)**
  - Подразумеваемое перемещение (по стандарту -- копирование);
  - Ссылка связывается только с lvalues,
    не позволяет перемещать из временных объектов.

- **T::T(T_ref)**
  - Подразумеваемое перемещение;
  - T_ref «связывается» и с lvalues, и с rvalues;

Новый вид ссылок:

T&&

Что к чему приводится

```
X x;
X& xlref = /*...*/;
const X& xclref = /*...*/;
X&& xrref = /*...*/;
```

```
X fx();
X& fxlref();
const X& fxclref();
X&& xrref();
```

	F(X&)	**F(const X&)**	**F(X&&)**
x xlref fxlref()	Да	Если нужно	static_cast
xclref fxclref()	Нет	Да	Нет
fx() fxrref()	Нет	Если нужно	Да
xrref	Да	Если нужно	static_cast

Что к чему приводится

```
X x;
X& xlref = /*...*/;
const X& xclref = /*...*/;
X&& xrref = /*...*/;
```

```
X fx();
X& fxlref();
const X& fxclref();
X&& xrref();
```

	~~F(X&)~~	F(const X&)	F(X&&)
x xlref fxlref()	~~Да~~	Да	static_cast
xclref fxclref()	~~Нет~~	Да	Нет
fx() **fxrref()**	~~Нет~~	Если нужно	Да
xrref	~~Да~~	Да	static_cast

Можно грабить ~~корованы~~ классы

```
template<class T>
bool has_duplicates(const std::vector<T>& v) {
    // make a copy and play with it
    std::vector<T> tmp = v;
    std::sort(tmp.begin(), tmp.end());
    return std::unique(tmp.begin(), tmp.end()) != tmp.end();
}

template<class T>
bool has_duplicates(std::vector<T>&& v) {
    // ‘v’ points to a temporary; no need to make a copy
    std::sort(v.begin(), v.end());
    return std::unique(v.begin(), v.end()) != v.end();
}
```

Перемещение классов

Перемещающие операции

- **T::T(const T&);**
  - копирующий конструктор;
- **T& T::operator = (const T&);**
  - копирующее присваивание;
- **T(T&&);**
  - перемещающий конструктор;
- **T& T::operator = (T&&);**
  - перемещающее присваивание.

Пример: std::vector

```
template<class T> class vector { // not really
private:
    T *start, *finish, *storage_end;
public:
    vector(const vector<T>&);                 // omitted
    vector& operator = (const vector<T>&); // omitted

    vector(vector<T>&& v):
        start(v.start), finish(v.finish),
        storage_end(v.storage_end)
    { v.start = v.finish = v.storage_end = 0; }

    vector<T>& operator = (vector<T>&& v) {
        std::swap(start, v.start);
        std::swap(finish, v.finish);
        std::swap(storage_end, v.storage_end);
    }
};
```

Время жизни объекта, из которого выполнили перемещение, не заканчивается!

Вспоминаем Майерса

«Если вашему классу нужен пользовательский конструктор копирования, оператор копирующего присваивания, конструктор перемещения, оператор перемещающего присваивания или деструктор -- скорее всего, ему нужно всё вышеперечисленное.»

Делать ли класс перемещаемым?

- Перемещающее присваивание можно сделать всегда;
  - достаточно обменять все поля класса;
- Перемещающий конструктор требует наличия пустого состояния;
  - оно моделируется пустым конструктором;
- Перемещающее присваивание без перемещающего конструктора выглядит странно;
- Если у класса нет допустимого пустого состояния –- проще целиком запретить перемещение.

Базис класса

- Класс перемещаемый
  - T::T();
  - T::T(const T&);
  - T& T::operator = (T&&);
  - T::T(T&& t): T()
    { *this = static_cast<T&&>(t); }
  - T& T::operator = (const T&)
    { return *this = T(t); }

- Класс неперемещаемый:
  - T::T(const T&);
  - void T::swap(T&);
  - T& T::operator = (const T&) {
      T(t).swap(*this);
      return *this;
    }

Автогенерация методов

- Если у класса нет недефолтных копирующих методов или деструктора -- компилятор сгенерирует перемещающие методы сам;
- Если у класса есть пользовательские перемещающие методы -- копирующие методы генерироваться не будут.
- Всегда можно попросить компилятор сгенерировать метод:
  - T(T&&) = default;
  - T& operator = (T&&) = default;

Заботливо разложенные грабли

```
class screen { // came from year 2004
private:
    std::vector< std::vector<char> > chars;

public:
    screen(): chars(25, std::vector<char>(80, '\x20')) {}

    void put_char(int x, int y, char c) {
        chars[y][x] = c;
    }
};
```

Если у класса есть инвариант,
не являющийся следствием инвариантов всех его полей и баз –
перемещение классу нужно писать руками!

Перемещение из переменных

```
void f(const X&); // #1
void f(X&&);      // #2

void g() {
  X x;

  f(x);                          // uses #1
  f(static_cast<X&&>(x));        // uses #2
  f(std::move(x));               // uses #2

  return x;    // will move
  throw x;     // will move
}
```

Два слова об эффективности

```
template<class T>
bool has_duplicates(const std::vector<T>& v) {
    std::vector<T> tmp = v;
    std::sort(tmp.begin(), tmp.end());
    return std::unique(tmp.begin(), tmp.end()) != tmp.end();
}

std::vector<int> read_file(const char* filename);

bool has_duplicates_in_file(const char* filename) {
    return has_duplicates(read_file(filename));
}
```

Два слова об эффективности

```
template<class T>
bool has_duplicates(std::vector<T> v) {
    std::sort(v.begin(), v.end());
    return std::unique(v.begin(), v.end()) != v.end();
}

std::vector<int> read_file(const char* filename);

bool has_duplicates_in_file(const char* filename) {
    return has_duplicates(read_file(filename));
}
```

Два слова об эффективности

- **Обещанные два слова:**

Не парьтесь!

- **Пишите то, что думаете:**
  - f(const X&) – «я только спросить»;
  - f(X) – «мне нужна локальная копия»;
- **Позвольте компилятору позаботиться об остальном.**

Переместить неперемещаемое

```
class X {
public:
    X();
    X(const X&);
};
void f(X);

void g() {
    X x;
    f(std::move(x));
}
```

- std::move – это разрешение компилятору выполнить перемещение;
- «Не можешь переместить? Скопируй!»

...и исключения

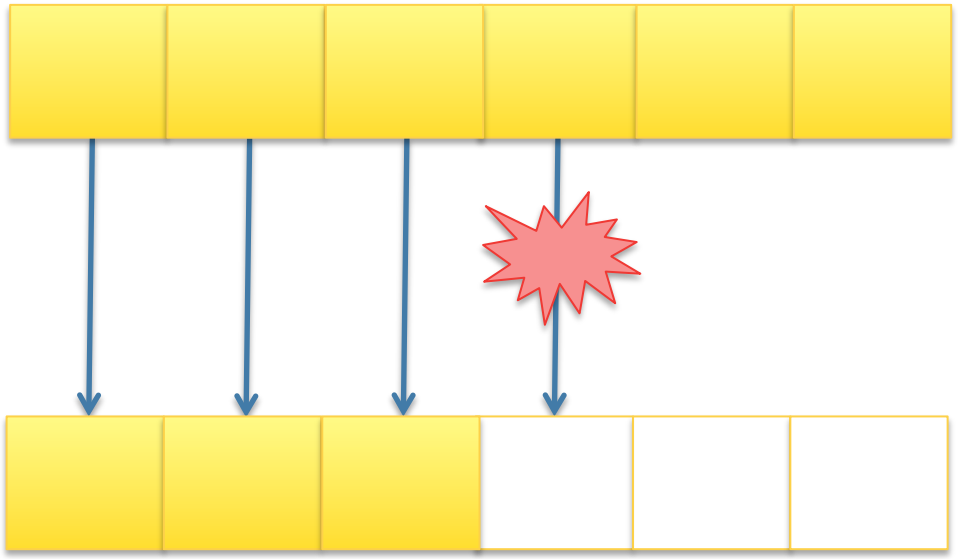
Снова std::vector

```
template<class T> class vector { // not really
private:
    T *start, *finish, *storage_end;

    void reallocate(size_t new_capacity);

public:
    size_t capacity() const { return storage_end - start; }
    void push_back(T t) {
        if (finish == storage_end) {
            reallocate((capacity() + 1) * 2);
        }
        new(finish) T(std::move(t));
        ++finish;
    }
};
```

Снова std::vector

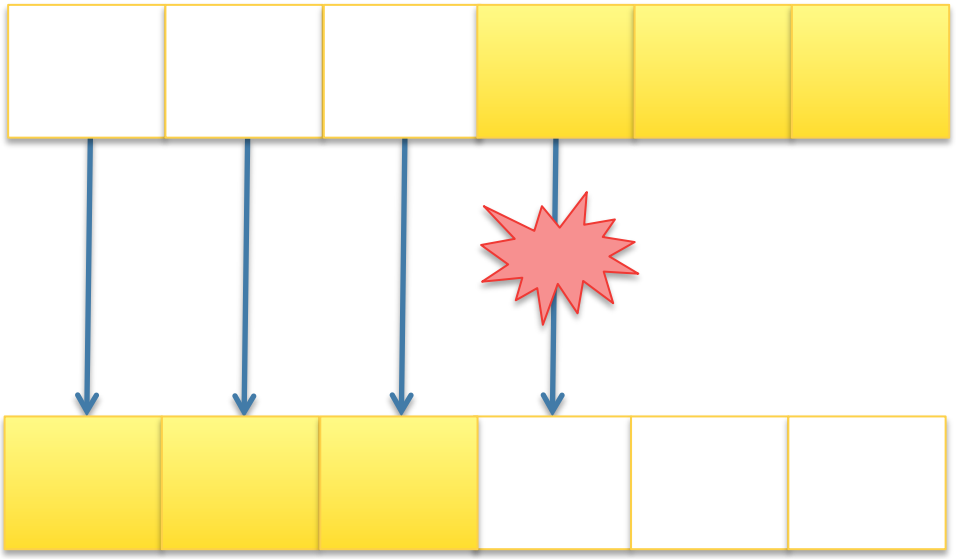
```
template<class T>
void vector<T>::reallocate(size_t new_capacity) {
    T* new_storage = malloc(new_capacity * sizeof(T));
    T* dest = new_storage;
    try {
        for (T* p = start; p != finish; ++p, ++dest)
            new(dest) T(std::move(*p));

        for (T* p = start; p != finish; ++p)
            p->~T();
        free(start);
        // apply things...
    }
    catch (...) {
        for (T* p = new_storage; p != dest; ++p)
            p->~T();
        free(new_storage);
        throw;
    }
}
```

Копируем несколько объектов



- Оригинальный набор объектов не изменяется;
- Тривиально обеспечить строгую гарантию.

Перемещаем несколько объектов



- Оригинальный набор объектов портится;
- Невозможно обеспечить гарантию лучше базовой;
  - Иногда даже базовую гарантию обеспечить не получается.

Перемещающие операции не ошибаются!

Но если очень хочется?

```
template<class X, class Y>
struct pair {
    X first;
    Y second;

    // compiler-generated
    pair(pair<X, Y>&& p):
        first(std::move(p.first)),
        second(std::move(p.second))
    {}
};
```

- Всё ли здесь в порядке?
- А если для Y не определены перемещающие операции?

Throw specifiers strike back!

Новое ключевое слово: **noexcept**

- f() noexcept – не может кидать исключений;
- f() noexcept(<cond>) – не может кидать исключений, если <cond> == true;
- noexcept(<expr>) – это bool, равный true, если вычисление <expr> не может кидать исключений.
  - ...то есть вызывает только операции над примитивными типами и функции, отмеченные как noexcept или throw().

...плюс пара trait-ов...

```
template<class T>
struct is_nothrow_move_constructible { // not really
    static bool value = noexcept(
        new(std:nothrow) T(std::move(*(T*)0)));
};

template<class T>
conditional<
    is_no_throw_move_constructible<T>::value,
    T&&, const T&
> move_if_noexcept(T&);
```

...и имеем строгую гарантию

```
template<class T>
void vector<T>::reallocate(size_t new_capacity) {

    // ...skipped...

        for (T* p = start; p != finish; ++p, ++dest)
            new(dest) T(std::move_if_noexcept(*p));

    // ...skipped...
}
```

Это не бесплатно

- Если f() noexcept всё-таки попробует выкинуть исключение – вызовется std::terminate();
  - компилятор вынужден оборачивать каждую такую функцию в try-block;
- Если компилятор не может доказать безопасность перемещения – он переключится на копирование.

Это не бесплатно

- Если вы пишете свои контейнеры -- не увлекайтесь!
  - «Строгая, кроме случаев, когда перемещающие операции могут генерировать исключения» -- вполне достаточная гарантия.
- Но перемещающие операции своих классов отмечать как noexcept всё-таки надо.
  - Сгенерированный компилятором метод сам получит такую метку, если нужно.

Универсальные ссылки

Reference collapsing

```
typedef cosnt X& XL;
typedef XL& XLL;
typedef volatile XLL& XLLL;
      // XLL = const volatile X&
```

- const ((volatile X&) &) & = const volatile X&

- (X&&) && = X&&
- (X&) && = (X&&) & = X&

...и выведение типов

```
template<class T> void f(T&&);

const X& cx = /*...*/; f(cx);  // T = const X&
X& lx = /*...*/; f(lx);        // T = X&
X x; f(x);                     // T = X&
f(std::move(x));               // T = X
X fx(); f(fx());               // T = X
```

T&& + выведение типа
= универсальная ссылка

Задача о перенаправлении*

- Дано: функция от одного аргумента и аргумент.
- Задача: вызвать функцию от этого аргумента.

```
template<class F, class Arg>
void call(F f, Arg arg) { f(arg); }
```

- Если f(x) компилируется – то call(f, x) тоже должна компилироваться и делать то же самое;
- Если f(x) не компилируется – то call(f, x) тоже не должна.
- Мы ничего не знаем ни про f, ни про x.

(*англ. *forwarding*)

Как принимать аргумент?

- void call(F f, Arg&);
  // нельзя вызывать с временными объектами;
- void call(F f, const Arg&);
  // не работает, если f() принимает неконстантную ссылку;
- void call(F f, const Arg&); void call(F f, Arg&);
  // а если у функции восемь аргументов?
- void call(F f, Arg arg);
  // молча всё ломает, если f() принимает Arg& и меняет его;
- void call(F f, const Arg& arg) { f(const_cast<Arg&>(arg); }
  // сегфолтится.

- void call(F f, Arg&& arg);
  // бинго!

Почти идеальный forwarding

```
template<class F, class Arg>
void call(F f, Arg&& arg) { f(arg); }
void gl(X&);
void gcl(const X&);
void gr(X);
```

Выражение	Выведенный тип Arg	Вид функции call()
X x; call(gl, x);	X&	call(F f, X& x) { f(x); }
const X& cx; call(gcl, cx);	const X&	X& x) { f(x); }
X fx(); call(gr, fx());	X	call(F f, X&& x) { f(x); }
X x; call(gr, std::move(x));	X	call(F f, X&& x) { f(x); }

Не хватает std::move()

Идеальный forwarding

```
template<class F, class Arg>
void call(F f, Arg&& arg)
{
    f(static_cast<Arg&&>(arg));
    f(std::forward<Arg>(arg));
}
```

Зачем это всё?

```
template<class T> class vector {
public:
    template<class... Args>
    void emplace_back(Args&&... args) {
        if (finish == storage_end)
            reallocate(capacity()*2);

        new(finish) T(std::forward<Args>(args)...);
        ++finish;
    }
};
```

std::forward() ≠ std::move()

```
template<class T>
typename remove_reference<T>::value&&
move(T&& t);
```

```
template<class T>
T&& forward(typename std::identity<T>::type&& t);
```

- move() всегда приводит свой аргумент к правой ссылке;
- forward() делает это, если только его шаблонный аргумент – не левая ссылка.

std::move() ≠ std::forward()

X x;	move(x)	X&&
	forward<X>(x)	X&&
const X& cx;	move(cx)	const X&& /*???*/
	forward<const X&>(cx)	const X&
X& lx;	move(lx)	X&&
	forward<X&>(lx)	X&
X&& rx;	move(rx)	X&&
	foward<X&&>(rx)	(не бывает)
	forward(x)	compile error

Спасибо за внимание!

Дмитрий Прокопцев

Яндекс.Карты,
старший разработчик

dprokoptsev@yandex-team.ru
http://github.com/dprokoptsev