

# Move семантика

Владислав Хорев  
Ведущий программист, Andersen



# Проверка связи





## Если у вас нет звука:

- убедитесь, что на вашем устройстве и на колонках включён звук
- обновите страницу вебинара (или закройте страницу и заново присоединитесь к вебинару)
- откройте вебинар в другом браузере
- перезагрузите компьютер (ноутбук) и заново попытайтесь зайти



## Поставьте в чат:

-  если меня видно и слышно
-  если нет

# Владислав Хорев

О спикере:

- Ведущий программист в компании Andersen
- Работает в IT с 2011 года
- Опыт разработки на C++ более 11 лет



# Вспоминаем прошрое занятие

**Вопрос:** какой указатель предоставляет  
разделяемое владение объектом?



# Вспоминаем прошрое занятие

**Вопрос:** какой указатель предоставляет  
разделяемое владение объектом?

**Ответ:** `std::shared_ptr`



# Вспоминаем прошрое занятие

**Вопрос:** какая проблема может возникнуть при использовании `std::shared_ptr`?



# Вспоминаем прошрое занятие

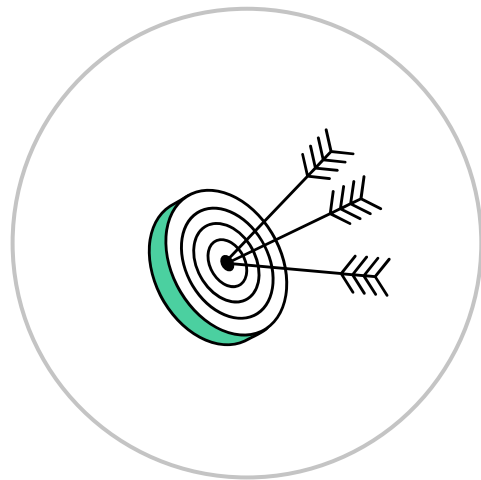
**Вопрос:** какая проблема может возникнуть при использовании `std::shared_ptr`?

**Ответ:** циклические ссылки



# Цели занятия

- Узнаем, что такое семантика перемещения
- Узнаем, как писать эффективный код, используя семантику перемещения

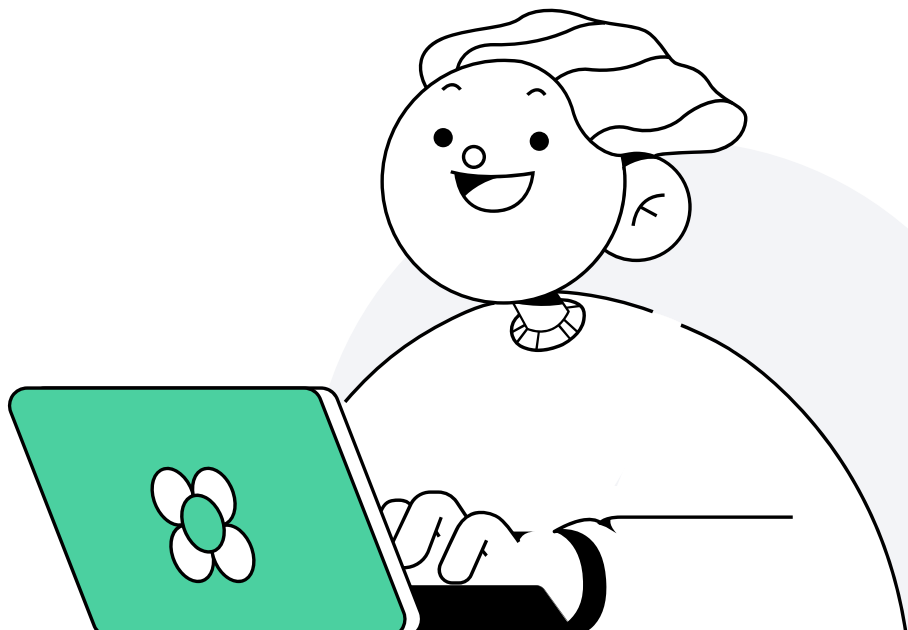




# План занятия

- 1 Перемещение объектов
- 2 Правило пяти (rule of five)
- 3 NRVO
- 4 Домашнее задание

\*Нажми на нужный раздел для перехода



# Перемещение объектов



1



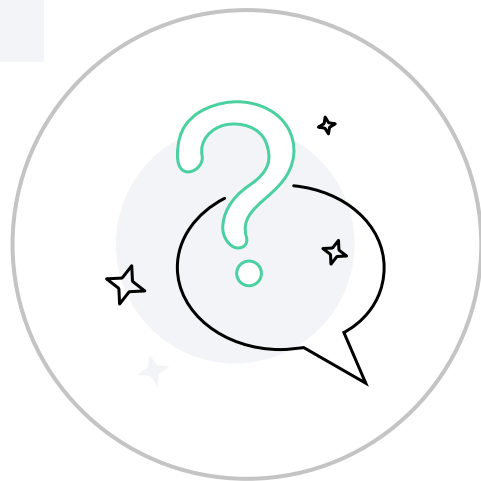
**Семантика перемещения - набор правил и средств языка C++, предназначенных для перемещения объектов, время жизни которых скоро истекает, вместо их копирования**

# Семантика перемещения

Рассмотрим функцию, которая принимает 2 массива и меняет их местами

```
void swap(std::vector<int>& lhs, std::vector<int>& rhs)
{
    std::vector<int> tmp = lhs;
    lhs = rhs;
    rhs = tmp;
}
```

**Вопрос:** Что не так с этой функцией ?



# Семантика перемещения

```
void swap(std::vector<int>& lhs, std::vector<int>& rhs)
{
    std::vector<int> tmp = lhs;
    lhs = rhs;
    rhs = tmp;
}
```

**Ответ:** Если передать в эту функцию большие массивы, например, по 1'000'000 элементов, то в силу особенностей реализации `std::vector` будет 3'000'000 лишних операций копирования



# Семантика перемещения

```
void swap(std::vector<int>& lhs, std::vector<int>& rhs)
{
    std::vector<int> tmp = std::move(lhs);
    lhs = std::move(rhs);
    rhs = std::move(tmp);
}
```

**Решение:** использовать `std::move`

Мы указываем компилятору, что эти объекты могут быть перемещены, и он преобразует эти объекты к специальной ссылке `rvalue`



# lvalue и rvalue

Если выражение ссылается на объект или можно взять адрес - то это **lvalue**. Все остальное - **rvalue**.

```
int a = 10;
a; // lvalue

int& b = a;
b; // lvalue, ссылается на a

void foo(int val)
{
    val; // lvalue
}

int& bar() { return a; }
bar(); // lvalue, ссылается на a

int bar() { return a; }
bar(); // rvalue
```

# lvalue и rvalue

std::move приводит lvalue к rvalue. Рассмотрим небольшой пример:

```
#include <iostream>

int x = 0;
int val() { return 0; }
int& ref() { return x; }

void test(int&) {
    std::cout << "lvalue\n";
}

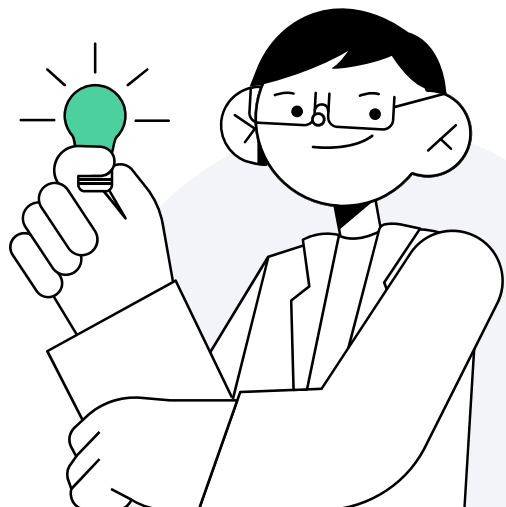
void test(int&&) {
    std::cout << "rvalue\n";
}

int main()
{
    test(0); // rvalue
    test(x); // lvalue
    test(val()); // rvalue
    test(ref()); // lvalue
    test(std::move(x)); // rvalue
    return 0;
}
```



# Когда перемещение не помогает?

- Если у объекта основные данные на стеке, то эти данные все равно придется копировать.
- `std::move` для константного объекта бесполезен.



# Правило пяти



2



**Если класс требует реализации одного из пяти следующих методов, то скорее всего, требует реализации всех пяти из них:**

- **деструктор**
- **конструктор копирования**
- **оператор копирующего присваивания**
- **конструктор перемещения**
- **оператор перемещающего присваивания**

```

class rule_of_five
{
    char* cstring; // сырой указатель
public:
    rule_of_five(const char* s = "") : cstring(nullptr)
    {
        if (s) {
            std::size_t n = std::strlen(s) + 1;
            cstring = new char[n];
            std::memcpy(cstring, s, n);
        }
    }

    ~rule_of_five() // деструктор
    {
        delete[] cstring;
    }

    rule_of_five(const rule_of_five& other) // конструктор копирования
    : rule_of_five(other.cstring) {}

    rule_of_five(rule_of_five&& other) noexcept // конструктор перемещения
    : cstring(std::exchange(other.cstring, nullptr)) {}

    rule_of_five& operator=(const rule_of_five& other) // оператор копирующего присваивания
    {
        return *this = rule_of_five(other);
    }

    rule_of_five& operator=(rule_of_five&& other) noexcept // оператор перемещающего присваивания
    {
        std::swap(cstring, other.cstring);
        return *this;
    }
}

```

Пример  
использования  
правила 5

# Правило пяти

Хочется обратить внимание на конструктор **rule\_of\_five(rule\_of\_five&& other)** и оператор перемещающего присваивания **rule\_of\_five& operator=(rule\_of\_five&& other)** Они принимают на вход rvalue.

Также можно в своих классах запрещать операцию обычного перемещения.

```
rule_of_five& operator=(const rule_of_five& other) = delete
```

Объект класса нельзя будет копировать, только перемещать

# NRVO



3



## Named return value optimization

**В некоторых случаях компилятор может (но это не гарантировано) самостоятельно использовать перемещение вместо ненужного копирования**

# NRVO

Допустим, есть следующая функция

```
my_class func() {  
    my_class local_variable;  
    // Действия с local_variable  
    return local_variable;  
}
```

И ее вызов

```
my_class result = func();
```

Казалось бы, должна быть следующая последовательность действий: вызов конструктора по умолчанию для `local_variable`, вызов оператора копирования, чтобы копировать `local_variable` в `result`, деструктор `local_variable`.



# NRVO

Допустим, есть следующая функция

```
my_class func() {  
    my_class local_variable;  
    // Действия с local_variable  
    return local_variable;  
}
```

И ее вызов

```
my_class result = func();
```

Однако компилятор оптимизирует код следующим образом, используя NRVO: он создаст result сразу в месте вызова функции. То есть не будет вызван оператор копирования и деструктор!

# NRVO

Условия для применения NRVO компилятором:

- Возвращаться должен именно локальный объект, а не ссылка или какая-то часть объекта
- Тип объекта, возвращаемого из функции согласно сигнатуре, должен совпадать с типом локального объекта

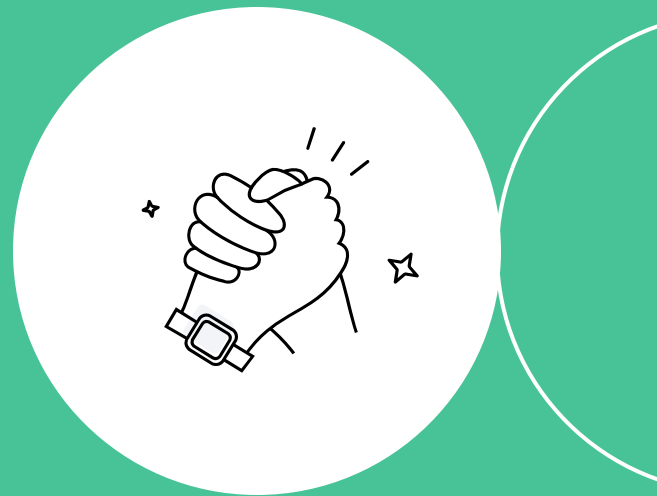
# RVO

Тоже самое, что NRVO. Оптимизирует конструкции вида:

```
my_class f() { return my_class (); }
```

В таких ситуациях компилятору легче выполнить оптимизацию, чем в случае с NRVO.

# Итоги



# Итоги занятия

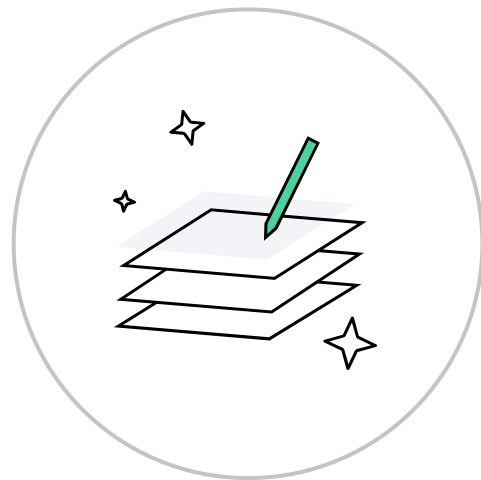
- 1 Познакомились с семантикой перемещения C++
- 2 Узнали какие оптимизации может использовать компилятор



# Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- 1 Вопросы по домашней работе задавайте в чате группы
- 2 Задачи можно сдавать по частям
- 3 Зачёт по домашней работе ставят после того, как приняты все задачи



# Дополнительные материалы

- [Статья про NRVO](#)
- [Статья про move семантику](#)



# Задавайте вопросы и пишите отзыв о лекции

Владислав Хорев  
Ведущий программист, Andersen

