

академия  
больших  
данных

mail.ru  
group

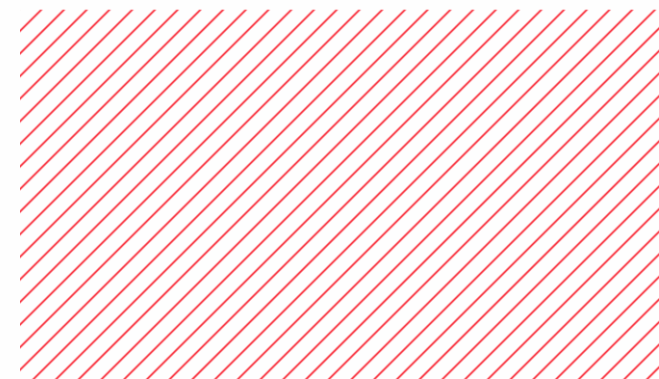


# Move-семантика

## Лекция 6

Гуров Роман

Advanced C++





# Проблема в C++03

---

```
std::vector<int> vec = std::vector<int>(100000);
```

Происходит лишнее копирование 100000 элементов

```
std::vector<int> vec(100000);
```

В данном случае можно просто писать так



# Проблема в C++03

---

```
std::vector<std::vector<int>> vec;  
vec.push_back(std::vector<int>(100000));
```

Опять приходится копировать временный объект,  
но теперь адекватной альтернативы нет



# Проблема в C++03

---

```
template <class T>
void swap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
```

По сути, хотим просто поменять местами “имена” объектов  
На деле – делаем три ненужных копирования



# Решение

---

В C++11 появилась “волшебная” функция `std::move`

```
template <class T>
void swap(T& a, T& b) {
    T temp = std::move(a);
    a = std::move(b);
    b = std::move(temp);
}
```

Говорим компилятору, что внутренности данного объекта можно “украсть” вместо копирования

# move-конструктор и оператор присваивания

---

На самом деле, для перемещения появились отдельные функции класса

```
struct C {  
    C(const C& other);  
    C& operator=(const C& other);  
    C(C&& other); ← конструктор перемещения  
    C& operator=(C&& other); ← move оператор присваивания  
};
```

&& означает ссылку на временный объект, который можно испортить (rvalue)



# Правило пяти

---

Вместо правила трёх, в C++11 действует правило пяти:

```
struct C {  
    C(const C& other);  
    C(C&& other);  
    C& operator=(const C& other);  
    C& operator=(C&& other);  
    ~C();  
};
```

Если хоть один из данных пяти методов реализован нетривиально,  
то скорее всего вам нужны все пять




# std::move

---

`std::move` просто преобразует тип объекта к временному

```
template <class T>  
typename remove_reference<T>::type&&
```

Хотим вернуть объект как временный  
Перед навешиванием rvalue-ссылки снимаем существующую







# std::move

---

`std::move` просто преобразует тип объекта к временному

```
template <class T>  
typename remove_reference<T>::type&& move(??? obj)
```

Непонятно, как можно принять





# std::move


---

`std::move` просто преобразует тип объекта к временному

```
template <class T>
typename remove_reference<T>::type&& move(??? obj) {
    return static_cast<typename remove_reference<T>::type&&>(obj);
}
```



Возвращаем просто результат каста к rvalue-ссылке



# rvalue-ссылки

---

rvalue-ссылки работают как обычные ссылки,  
но привязываются только к rvalue

```
int a = 5;  
int&& b = a; // Ошибка компиляции  
int&& c = 5;  
int& d = c;  
int&& e = c; // Ошибка компиляции  
int&& f = std::move(c);  
const int& g = 5;
```

Нельзя привязать **const** имя к не-**const** ссылке



# Виды value

---

## Примеры lvalue

```
int* x;
```

```
x;
```

```
++x;
```

```
x = y;
```

```
*x;
```

## Примеры rvalue

```
int x;
```

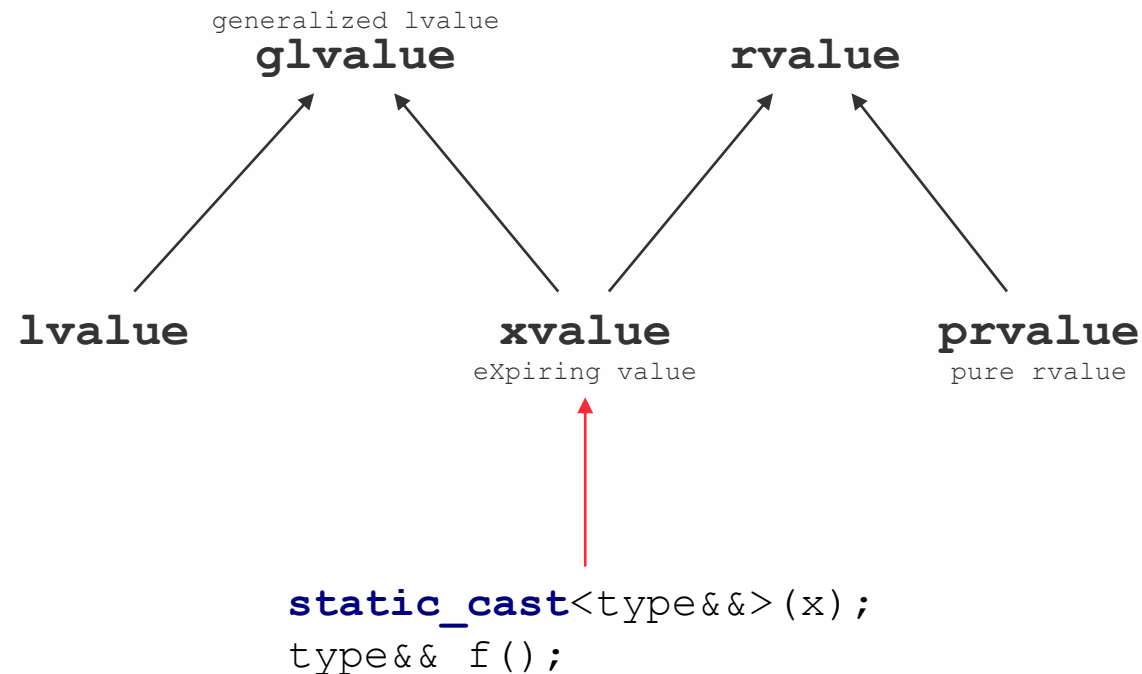
```
x++;
```

```
5;
```

```
std::vector<int>();
```

От lvalue объектов можно взять указатель,  
от rvalue – нельзя

# Виды value



**xvalue** – именованные объекты, ресурсы которых можно переиспользовать.

Посмотреть список всех выражений того или иного вида можно на [cppreference.com](http://cppreference.com)

# Универсальные ссылки

Для подстановки `T&&` в C++11 сделано особое правило

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

int main() {
    int x;
    f(x);
    f(5);
}
```

T&&	T
f(x)	<b>int&amp;</b>
f(5)	<b>int</b>

Но что будет с x?



# Правило сворачивания ссылок

---

Если при подстановке шаблонных типов попадают  
две ссылки подряд, то lvalue-ссылка побеждает

```
type&& & -> type&
type& && -> type&
type&& && -> type&&
```

# Универсальные ссылки

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

int main() {
    int x;
    f(x);
    f(5);
}
```

T&&	T	x
f(x)	int&	int& && -> int&
f(5)	int	int && -> int&&

В итоге, функция `f` может принимать и rvalue, и lvalue





# std::move

---

Теперь можем использовать универсальную ссылку  
для реализации `std::move`

```
template <class T>
typename remove_reference<T>::type&& move(T&& obj) {
    return static_cast<typename remove_reference<T>::type&&>(obj);
}
```

`remove_reference` нужен в случае передачи lvalue



# vector::push\_back

---

Можно решить проблему с `push_back`, сделав перегрузку для rvalue аргументов

```
void push_back(const T& x);  
void push_back(T&& x);
```

```
std::vector<std::vector<int>> vec;  
vec.push_back(std::vector<int>(100000));
```

Тот же самый код теперь работает быстрее

# Прямая передача

---

```
template <class T>
void push_back_and_log(T&& x) {
    std::cout << x << std::endl;
    push_back(x); ← В теле функции x уже является lvalue
}
```

Хочется как-то вызывать правильную версию `push_back`,  
в зависимости от того, что нам изначально передали

# std::forward

---

Для прямой передачи значений, принятых по универсальной ссылке, существует функция `std::forward`

```
template <class T>
void push_back_and_log(T&& x) {
    std::cout << x << std::endl;
    push_back(std::forward<T>(x));
}
```

Сюда передается разный тип, в зависимости от того, что приняли

Теперь будет вызвана правильная перегрузка метода `push_back`, в зависимости от того, что передали в `push_back_and_log`

# Реализация std::forward

---

```
std::forward<T>(x)
```

```
template <class T>
T&& forward(typename remove_reference<T>::type& x) {
    return static_cast<T&&>(x);
}
```

Получается, что для `T&` функция будет просто возвращать по ссылке,  
а для `T` – сделает то же, что и `std::move`



# construct и emplace\_back

---

```
template<class T, class... Args>
void construct(T* p, Args&&... args) {
    new (p) T(args...);
}
```

```
template<class... Args>
void emplace_back(Args&&... args) {
    // ...
    construct(ptr, args...);
    // ...
}
```

Тут уже по-настоящему пригождаются универсальные ссылки, ведь аргументы могут быть rvalue и lvalue вперемешку



# construct и emplace\_back

---

```
template<class T, class... Args>
void construct(T* p, Args&&... args) {
    new (p) T(std::forward<Args>(args)...);
}
```

```
template<class... Args>
void emplace_back(Args&&... args) {
    // ...
    construct(ptr, std::forward<Args>(args)...);
    // ...
}
```

И `std::forward` позволяет передать аргументы оптимально



# Новая проблема с push\_back

---

Если при добавлении элемента понадобилось выделить память большего размера, нужно перемещать элементы из старой памяти в новую

При этом хочется соблюдать строгую гарантию безопасности при исключениях

Но если во время перемещения бросится исключение, то и старая и новая память останутся в невалидном состоянии

Как быть?





# std::move\_if\_noexcept

---

На такие случаи есть специальный вариант `std::move`

```
std::move_if_noexcept(x);
```

Он возвращает rvalue-ссылку, только если  
конструктор перемещения у `x` обозначен **noexcept**



# Return Value Optimization

---

```
std::string operator+(const std::string& x) {  
    std::string ans = *this;  
    ans += x;  
    return ans;  
}
```

Нужно ли писать `std::move` в `return`?

Нет, компилятор умеет делать возврат по значению из функции даже без конструирования нового объекта



# Return Value Optimization

---

Не всегда у компилятора есть возможность сделать RVO

```
if (...) {  
    return str1;  
} else {  
    return str2;  
}
```

В таком случае стандарт обещает, что будет вызвано перемещение.



# Return Value Optimization

---

RVO не произойдет, если явно написать `std::move`

```
std::string operator+(const std::string& x) {  
    std::string ans = *this;  
    ans += x;  
    return std::move(ans);  
}
```

Поэтому, писать так не стоит



# Copy Elision

---

Подобные оптимизации возможны не только с возвращаемым объектом

```
T f() {  
    return T();  
}  
  
T x = T(T(f()));
```

Стандарт C++17 гарантирует, что в данном случае будет вызван только конструктор по умолчанию, сразу инициализируя `x`



# reference qualifiers

---

```
std::string operator+(const std::string& x) {  
    std::string ans = *this;  
    ans += x;  
    return ans;  
}
```

В данной ситуации ничто не запрещает написать такой код:

```
a + b = c;
```

Хотелось бы сделать так, чтобы оператор присваивания не работал с rvalue



# reference qualifiers

---

Можно использовать ссылочные квалификаторы, чтобы указать, что данный метод можно вызвать только у lvalue или rvalue объекта

```
class C {  
    void f() & {};  
    void f() && {};  
};
```