

C++

Fall 2020

compscicenter.ru

Башарин Егор, t.me/egorbasharin

Филипп Грабовой, t.me/phil_grab

Лекция XI

Value Categories, Move Semantics, Perfect Forwarding

Section 1

категории значений `glvalue` и `prvalue`

Motivation

Наличие категорий значений выражений позволяет создавать быстрые программы:

- за счет переиспользования ресурсов временных или перемещаемых объектов
- за счет оптимизаций компилятора (например, copy elision)

Expression & value category

В C++ у выражений есть свойство `value category`.

Любое выражение имеет либо категорию `glvalue`, либо `prvalue`.

glvalue & prvalue

Результат glvalue выражения — **ссылка на объект**¹.

Результат prvalue выражения — **значение**², либо имеет тип `void`.

(1) Раз есть ссылка, значит и участок памяти, где располагается объект, фиксирован.

(2) Значение можно представить как объект, память для которого не фиксирована и представляет некое временное хранилище для него.

объекты и значения

Создание объекта подразумевает выделение памяти для него.

Если эта память фиксирована ("связана с объектом"), то можно использовать ссылки и указатели на этот объект.

объекты и значения

Пример

```
int i = 2;
```

Создание переменной выделяет память и "связывает" ее с **объектом** типа `int`.

Для доступа к **объекту**, используется имя `i`.

Литерал `2` — это **значение**, используемое для инициализации объекта, на который ссылается `i`.

glvalue & prvalue

```
int i, j;  
  
i = (1 + 2);  
glvalue j = (i + 5);  
  
(i > j ? i : j) = 10;  
  
(1 + 2) = 100; // Error
```

prvalue

glvalue & prvalue

```
struct T { int i; };  
T makeT() { return T{1}; }
```

```
T t{2};  
T& globalT() { return t; }
```

```
int main() {  
    T t{10};  
    t = makeT();  
    t = globalT();  
}
```

Section 2

lvalue, xvalue, rvalue, rvalue-references

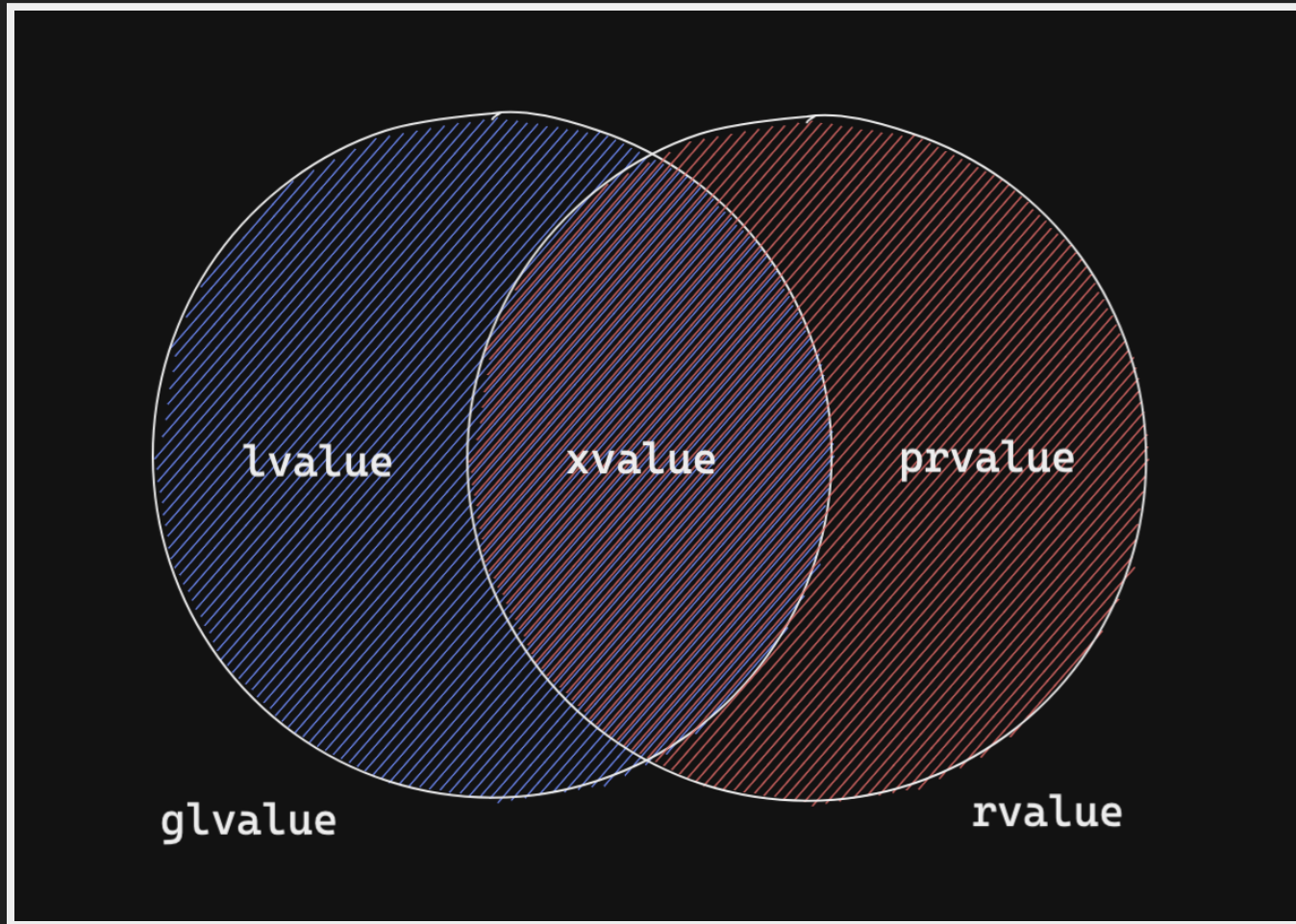
Value Categories

xvalue-выражение — это glvalue-выражение, чей результат ссылается на объект, у которого можно переиспользовать ресурсы

lvalue — это glvalue за исключением xvalue

rvalue — это объединение xvalue и prvalue

Value Categories



Этимология

rvalue & lvalue

Так исторически сложилось, что r и l указывают на то, с какой стороны от оператора присваивания могут быть использованы эти выражения.

Но, это не всегда так, поэтому возникает путаница:

ЭТИМОЛОГИЯ

glvalue, prvalue, xvalue

- glvalue — "generalized" lvalue
- prvalue — "pure" rvalue
- xvalue — expiring value

rvalue-references

Пример lvalue-ссылок:

```
int main() {  
    int i = 1;  
    int& lvalueRef1 = i;  
    const int& lvalueRef2 = i;  
    const int& lvalueRef3 = 10; // extends lifetime  
}
```

Для rvalue-ссылок используется двойной амперсанд, а инициализаторами могут быть только rvalue-выражения¹:

```
int main() {  
    int i = 1;  
    int&& rvalueRef1 = i; // compile-time error  
    int&& rvalueRef2 = 10; // OK, extends lifetime  
}
```

Здесь и далее в секции будем рассматривать rvalue-ссылки на `int` только для того, чтобы показать синтаксис и примеры работы, полезный случай использования разберем в следующей секции.

(1) Не все `rvalue`-выражения подходят. Пример: вызов функции, которая возвращает void.

rvalue-references

```
int&& rvalueRef = 10; // OK, extends lifetime
```

10 — это prvalue-выражение, а значит и rvalue-выражение.

rvalueRef может ссылаться на объект, который "связан" с памятью: поэтому создается временный объект типа `int` и инициализируется значением 10.

rvalue-references

Выражение rvalueRef — это lvalue-выражение

```
int&& rvalueRef = 10;    // OK, extends lifetime  
int&& rvalueRef2 = rvalueRef; // Error
```

Можно использовать static_cast:

```
int&& rvalueRef2 = static_cast<int&&>(rvalueRef);
```

Инициализатор — xvalue-выражение

xvalue

Для получения xvalue-выражения можно привести lvalue-выражение к rvalue-ссылке:

```
int i = 10;  
int&& rvalueRef3 = static_cast<int&&>(i);
```

rvalue-references & functions

```
int f(const int&) { return 1; }
int f(int&&) { return 2; }
int&& g(int&& x) { return static_cast<int&&>(x); }

int main() {
    int i = 10;
    assert(f(i) == 1); // pass lvalue-expr as arg
    assert(f(1) == 2); // pass prvalue-expr as arg

    // pass xvalue-expr as arg
    assert(f(static_cast<int&&>(i)) == 2);
    assert(f(g(1)) == 2);
}
```

Перегрузка функции выбирается в зависимости от категории значения аргумента.

Вызов функции, возвращающей rvalue-ссылку¹ — еще один способ получить xvalue-выражение.

(1) Не возвращайте ссылки на объекты, созданные в теле функции (dangling reference)!!!

rvalue-references & functions

```
int&& g(int&& x) {  
    return static_cast<int&&>(x);  
}
```

Тип параметра функции требует, чтобы аргументом было rvalue-выражение.

Выражение `x` внутри функции — это **lvalue-выражение**, чтобы получить rvalue, следует воспользоваться `static_cast`.

Section 3

move-semantics

Пример 1

```
template <class T> struct Vector {
    Vector() = default;
    Vector& operator=(const Vector& o);
private:
    value_type* buf_ = nullptr;
    size_type size_ = 0;
    size_type cap_ = 0;
};

Vector<T> createAnyVector();

int main() {
    Vector<T> v;
    v = createAnyVector();
}
```

Какие здесь проблемы?

Пример 1

Пусть результат функции `createAnyVector` это вектор TMR.

После выполнения этого утверждения:

```
v = createAnyVector();
```

вектор TMR будет уничтожен: **вызов деструкторов у всех элементов вектора и деаллокация.**

Но TMR передается в `operator=` по константной ссылке, поэтому придется **аллоцировать** память `buf_` и **копировать** объекты из `o.buf_`.

Как хотелось бы: раз объект "временный", то он никому не нужен — заберем его ресурсы и избавимся от **лишней работы.**

Пример 1

Fix:

```
template <class T> struct Vector {
    Vector() = default;
    Vector& operator=(const Vector& o);
    Vector& operator=(Vector&& o);
private:
    value_type* buf_ = nullptr;
    size_type size_ = 0;
    size_type cap_ = 0;
};

Vector<T> createAnyVector();

int main() {
    Vector<T> v;
    v = createAnyVector();
}
```

Пример 1

Теперь будет выбран `operator=(Vector&& o)`, так как выражение `createAnyVector()` имеет категорию `rvalue`.

Если параметр функции является `rvalue`-ссылкой, то можно использовать (заимствовать) ресурсы аргумента, который соответствует этому параметру.

Source code: [Click me](#)

Пример 2

```
void someFunc(Vector<T>&& vec) {  
    Vector<T> v;  
    v = vec;  
}
```

Пример синтетический. Можно было бы сразу инициализировать `v` вектором `vec`, но мы рассматриваем работу с разными перегрузками оператора=

Пример 2

```
void someFunc(Vector<T>&& vec) {  
    Vector<T> v;  
    v = vec;  
}
```

Выражение справа от знака = имеет категорию `lvalue`, поэтому вызовется `const ref` перегрузка оператора.

Но, нам известно, что `vec` "временный", так как параметр — это `rvalue`-ссылка.

Нужно явно указать, что справа от знака = `rvalue`-выражение:

```
v = static_cast<Vector<T>&&>(vec); // xvalue
```

Пример 2

`std::move` from `<utility>`

```
#include <utility>

void someFunc(Vector<T>&& vec) {
    Vector<T> v;
    v = std::move(vec); // same as static_cast<Vector<T>
}
```

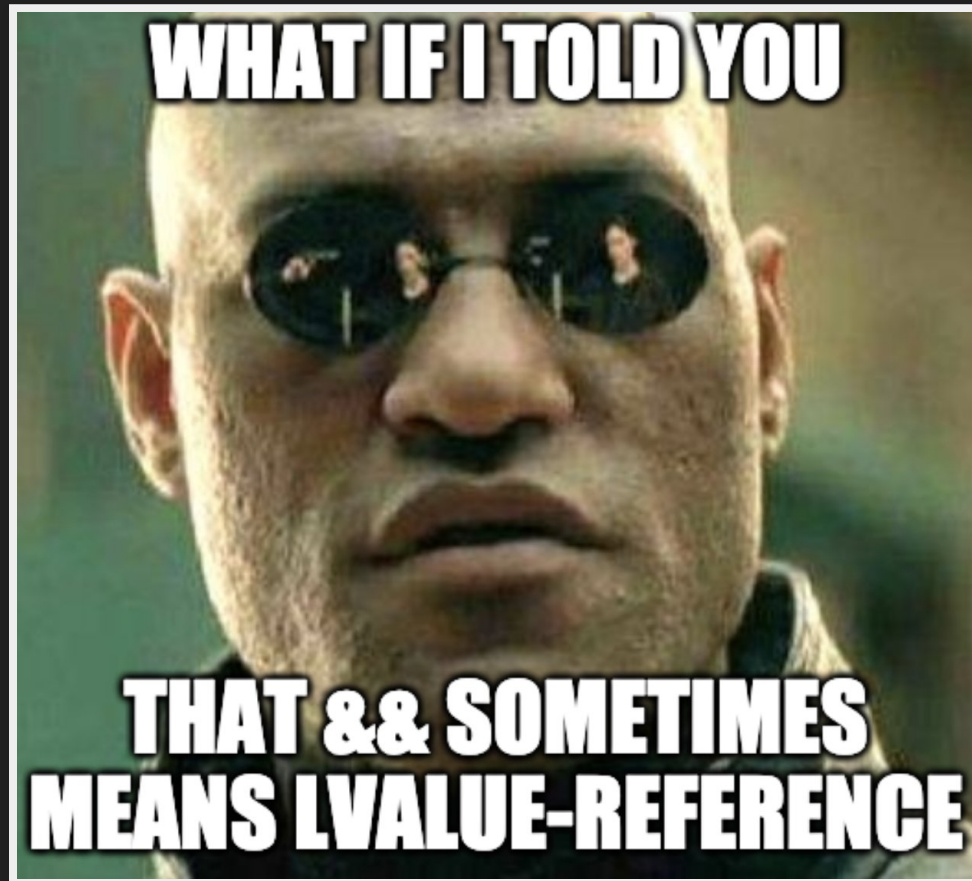
Чтобы понять как устроен `std::move`, нужно разобраться с темой универсальных ссылок

move-semantics

- move-семантика позволяет избежать ненужных вычислений, тем самым улучшает производительность программы
- пользуемся тем, что можем забрать ресурсы из "временных" объектов
- для того чтобы различать "временность" объекта, заводим две перегрузки функции: с lvalue ссылкой и rvalue ссылкой.
- rvalue ссылка лишь требует, чтобы ее инициализировали временным объектом, выражение состоящее из имени ссылки имеет категорию lvalue
- чтобы преобразовать lvalue в rvalue используем `std::move`

Section 4

universal references



defining universal references

```
template <class T>
void f(T&& t) { }           // t -- universal reference

struct S {
    template <class T>
    void g(T&& t) { }       // t -- universal reference
};
```

Параметр шаблонной функции, объявленный как rvalue-ссылка на параметр шаблона **этой функции**.

Для примера выше:

- * `t` -- параметр шаблонной функции
- * `T&& t` -- объявление параметра функции
- * `T` -- параметр шаблона
- * `T&&` -- rvalue-ссылка на параметр шаблона

usage example

```
template <class T>
void f(T&& t) { }

// 1.
f(10);           // rvalue -> [T = int] -> [T&& = int&&

// 2.
int i = 10;
f(i);           // lvalue -> [T = int&] -> [T&& = int&

// 3.
f(std::move(i)); // rvalue -> [T = int] -> [T&& = int&&
```

<https://cppinsights.io/s/e652eac4>

Первая стрелка показывает результат вывода типа
Вторая стрелка показывает результат схлопывания ссылок

Reference collapsing

Схлопывание ссылок

Проблема ссылки на ссылку при использовании псевдонимов типов и параметров шаблона.

	[T = U&]	[T = U&&]
T&	U&	U&
T&&	U&	U&&

<https://cppinsights.io/s/9814f738>

Finding universal reference

```
void f(int&& r) { } // (1)
```

```
template <class T>  
void g(T&& t) { } // (2)
```

```
template <class T>  
void h(const T&& t) { } // (3)
```

```
template <class T>  
void p(std::vector<T>&& v) { } // (4)
```

Finding universal reference

```
template <class T>
struct S {

    void f(T&& t) {}    // (1)

    template <class U>
    void g(U&& u) {}    // (2)

};
```

unveil `std::move`

Чтобы понять, как работает `std::move`, напомним свой:

Заготовка: [Click me](#)

Решение: [Click me](#)

Section 5

perfect-forwarding

Example

```
struct Object {};  
int processImpl(const Object&) { return 1; }  
int processImpl(Object&&) { return 2; }  
  
template <class T>  
int process(T&& t) {  
    return processImpl(t);  
}  
  
int main() {  
    Object obj;  
    assert(process(obj) == 1);  
    assert(process(std::move(obj)) == 2);  
    assert(process(Object{}) == 2);  
}
```

Example

```
...  
template <class T>  
int process(T&& t) {  
    return processImpl(t);  
}  
...
```

Всегда будет выбираться перегрузка с параметром lvalue-ссылкой, так как выражение `t` имеет категорию `lvalue`

Как починить?

Fixing...

```
...  
template <class T>  
int process(T&& t) {  
    return processImpl(t);  
}  
...
```

Что хотелось бы сделать:

- T — ссылка:
 - lvalue: processImpl(t)
 - rvalue: processImpl(std::move(t))
- T — не ссылка: processImpl(std::move(t))

Fixed

`std::forward` from `<utility>`

```
...  
template <class T>  
int process(T&& t) {  
    return processImpl(std::forward<T>(t));  
}  
...
```

unveil std :: forward

Чтобы понять, как работает `std :: forward`, напишем свой:

Заготовка: [Click me](#)

Решение: [Click me](#)