

Семестр 2. Лекция 6. C++11.

Евгений Линский

21 Апреля 2017

- ▶ Major: c++98, c++11, c++17
 - c++11 — поддерживается компиляторами
 - c++17 — draft stage в марте 2017; после draft не будет серьезных изменений; выйдет в финальной версии в конце года.
- ▶ Minor: c++03, c++14 (например, bug fixes)
- ▶ gcc: `-std=c++11`

Explicitly defaulted and deleted special member functions

Было:

```
class A {  
public:  
    A(int);  
    A() {}; // 1.  автоматически не сгенерируется из-за A(int)  
            // 2.  нужен для array = new A [100]  
private:  
    // сделать приватным --> запретить копирование  
    A(const A&);  
    A& operator=(const A&);  
};
```

Приватные copy constructor и assing operator:

- ▶ Использование: Singleton, scoped_ptr
- ▶ Проблемы: все-таки можно копировать в friend классе или в самом классе

Explicitly (явно!) defaulted and deleted special member functions

В C++11 стало явно:

```
class A {  
public:  
    A(int);  
    A() = default;  
    A(const A&) = delete;  
    A& operator=(const A&) = delete;  
};
```

Explicit overrides and final

Было:

```
class Base {  
public:  
    virtual void f(int);  
    virtual int g() const;  
    void h(int);  
};
```

```
class Derived: public Base {  
public:  
    void f(int);  
    int g();  
    void h(int);  
};
```

Проблемы:

- ▶ чтобы понять является ли `f` виртуальной, надо смотреть в базовый класс.
- ▶ для функции `g` случайно сделали перегрузку, вместо перекрытия.
- ▶ для функция `h` тоже не перекрывается.

Explicit overrides and final

В C++11 стало явно:

```
class Derived: public Base {  
public:  
    void f(int) override; // ok  
    int g() override; // compilation error  
    void h(int) override; // compilation error  
};
```

Explicit overrides and final

```
struct Base {  
    virtual void f();  
};  
struct Derived : public base {  
    void f() final;    // virtual as it overrides base::f  
    void g() { f(); } // direct call without virt. func.  
    table (optimization!)  
};  
struct mostderived : public derived {  
    void f(); // error: cannot override!  
};
```

```
struct Base1 final { };  
struct Derived1 : Base1 { }; // error
```

Object construction improvement

```
C++11:
class A {
    int avg = 1; // initial value
    A(int a1, a2) { avg = a1 + a2; }
    A(int *array) : A(array[0], array[1]) { } //constructor
    chaining
};
```

```
struct Base1 final { };
struct Derived1 : Base1 { }; // error
```

Initializer lists

Было:

```
struct point {  
    int x;  
    int y;  
};  
point p = {100, 100};  
int a[] = {10, 20, 30};
```

Стало C++11:

```
std::vector<std::string> v = { "AA", "AB", "AC" };  
std::vector<std::string> v({ "AA", "AB", "AC" });  
std::vector<std::string> v{ "AA", "AB", "AC" };
```

Initializer lists

```
template <typename T>
class vector {
public:
    SequenceClass(std::initializer_list<string> list) {
        const T *array = l.begin();
        T value = *(array++);
        ...
        l.size();
    }
};
```

Можно передавать `initializer_list` без ссылки:

- ▶ там “скорей всего” внутри всего лишь 2 указателя
- ▶ кроме того, вызовется `move` (будет объяснено дальше), а не `copy`

- ▶ lvalue — может быть и в левой, и в правой части присваивания (переменные, включая константы)
 - продолжает существовать за пределами выражения, где было использовано
 - есть имя
 - можно взять адрес
- ▶ rvalue — выражение, которое может быть только в правой части присваивания
 - не существует за пределами выражения, где было использовано
 - временное значение (temporary value)

Примеры:

```
int a = 42;
int b = 43;
// a * b is an rvalue:
int c = a * b; // ok, rvalue on right hand side of
assignment
a * b = 42; // error, rvalue on left hand side of
assignment
```

```
vector<Person> people;
people.push_back(Person("Evgeny", 36)); //
Person("Evgeny", 36) -- rvalue
```

```
int square(int x) { return x*x; }
int sq = square(10); // square(10) is an rvalue
```

Пусть X класс, который:

- ▶ в своих полях хранит ресурсы: указатель на динамически выделенную память, файлы и т.п.

```
class X{  
    int *array;  
    X(size_t size);    ~X();  
    X(const X&); X& operator=(const X&);  
};
```

Накладные расходы!

```
X foo();  
X x;  
x = foo(); // 1. клонировать ресурс из temporary  
           // 2. освободить ресурс в x (delete, fclose, etc)  
           // 3. присвоить в x клонированный ресурс  
           // 4. освободить ресурс в temporary
```

Хочу эффективно обработать эту ситуацию!

```
class X{
    int *array;
    X(size_t size);   ~X();
    X(const X& x); X& operator=(const X& x);
    X(X&& x) { std::swap(this->array, x.array); }
    X& operator=(X&& x) { ... }
};
```

```
X x1(100);
X x2(x1); //call X(const X& x), т.к.  x1 -- lvalue

X x3(X(100)); //cal X(X&& x), т.к.  X(100) -- rvalue
```

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

Вызовутся ли move constructor и move assignment operator?

std::move —“cast” из lvalue reference в rvalue reference

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