Объектно-ориентированное программирование

Object-oriented programming

VI. Жизнь и смерть объектов

Resource Acquisition Is Initialization

Правила конструирования объектов

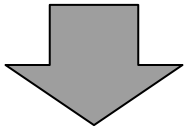
compiler implicitly declares

user declares	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

http://howardhinnant.github.io/classdecl.html

Категории значений* (l-values/r-values)

```
void f(int&);
void g(int&&);
void h(const int&);
```



```
int i = 0;
f(i);
g(i); // an rvalue reference cannot be bound to an lvalue
h(i);
```

```
f(42); // initial value of reference to non-const must be an lvalue
g(42);
h(42);
```

Первое правило (C.21)

"If you **define** or **=delete** *any* **copy**, **move**, or **destructor** function, **define** or **=delete** *them all*."

B. Stroustrup

"CppCoreGuidelines"

https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Rc-five

Вопрос на засыпку

```
class vector {
public:
    template <class ...Args> vector(Args&& ...args);
private:
    int *p{nullptr};
    int *n{nullptr};
    int *cap{nullptr};
};
```

Variadic template constructor

```
class vector {
public:
    template <class ...Args> vector(Args&& ...args);
    vector() = delete;
private:
    int *p{nullptr};
    int *n{nullptr};
    int *cap{nullptr};
};
```

Правило “трех” (rule of three)

```
class vector {
public:
    vector() = default;
    vector(size_t len) :
        p(new int[len]), n(p + len), cap(n) {}
private:
    int *p{nullptr};
    int *n{nullptr};
    int *cap{nullptr};
};
```

Правило “трех”

```
try {
    vector a;
    vector b(4);  // memory leak
    vector c = a; // double free
    c = b;        // memory leak
} catch (const std::exception& e) {
    std::cerr << e.what() << '\n';
}
```

Правило “трех”

```
class vector {
public:
    // …
    ~vector() { delete[] p; }
    vector(const vector &other) :
        p(new int[other.n - other.p]),
        n(p + (other.n - other.p)),
        cap(n)
    {
        std::copy(other.p, other.n, p);
    }
```

```
    vector& operator=(
        const vector &other)
    {
        if(&other != this) {
            delete[] p;
            p = new int[
                other.n - other.p];
            n = p + (other.n - other.p);
            cap = n;
            std::copy(
                other.p, other.n, p);
        }
        return *this;
    }
    // …
};
```

“Умные” указатели (smart pointers)

```
try {
    vector *d = new vector;
    // throw here
    delete d;                    // memory leak
} catch(const std::exception& e) {
    std::cerr << e.what() << '\n';
}
```

https://en.cppreference.com/w/cpp/memory

“Умные” указатели

```
class unique_ptr {
public:
    ~unique_ptr() { delete[] p; }
    unique_ptr(int *ptr) : p(ptr) {}
    unique_ptr() = delete;
    unique_ptr(const unique_ptr&) = delete;
    unique_ptr& operator=(
        const unique_ptr&) = delete;
private:
    int *p{nullptr};
};
```

“Умные” указатели

```
try {
    unique_ptr p = new int[4];
    // throw here
} catch(const std::exception& e) {
    std::cerr << e.what() << '\n';
}
```

Правило “нуля” (rule of zero)

“Code that is ***not*** **written** cannot be wrong.”

P. Sommerlad

“Introducing the rule of DesDeMovA”, 2019

https://safecpp.com/2019/07/01/initial.html

Правило “нуля”

```
class queue {
public:
    ~queue() = default;
    queue() = default;
    queue(const queue&) = default;
    queue& operator=(const queue&) = default;
    void push(int v) {
        data.push_back(v);
    }
private:
    std::vector<int> data;
};
```

Правило “нуля”

```
try {
    queue p;
    p.push(1);
    // throw here
} catch(const std::exception& e) {
    std::cerr << e.what() << '\n';
}
```

“Telling a programmer there's already a library to do "*X*" is like telling a songwriter there's already a song about love.”

P. Cordell

Категории значений при перегрузке функций

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

int main(int argc, char const *argv[])
{
    vector a = {1, 2, 3, 4};
    f(a);
    f({1, 2, 3, 4});
    f(std::move(a));
}
```

Категории значений при перегрузке функций

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

int main(int argc, char const *argv[])
{
    vector a = {1, 2, 3, 4};
    f(a);              // #1
    f({1, 2, 3, 4}); // #2
    f(std::move(a)); // #2
}
```

std::move (и синтезированный конструктор переноса)

```
template <typename _Tp>
constexpr typename std::remove_reference<_Tp>::type&&
move(_Tp&& __t) noexcept {
    return
       static_cast<typename std::remove_reference<_Tp>::type&&>(__t);
}
/* Convert a value to an rvalue.
   Parameters:
   __t - A thing of arbitrary type.
   Returns:
   The parameter cast to an rvalue-reference to allow moving it. */
```

Правило “пяти” (rule of five)

```
class vector {
public:
    // …
    vector& operator=(
        const vector &other)
    {
        if(&other != this) {
            // copy-and-swap:
            vector tmp(other);
            tmp.swap(*this);
        }
        return *this;
    }
    vector(
        vector &&other) noexcept :
        p(std::exchange(other.p, nullptr)),
        n(std::exchange(other.n, nullptr)),
        cap(std::exchange(
            other.cap, nullptr)) {}
    vector& operator=(
        vector &&other) noexcept
    {
        vector tmp(std::move(other));
        tmp.swap(*this);
        return *this;
    }
    // …
};
```

Правило “четырех с половиной”

```
class vector {
    unique_ptr<int[]> p;
public:
    ~vector() = default;
    vector() = default;
    vector(const vector &other) :
        p(make_unique<int[]>(
            other.n - other.p.get())),
        n(p.get() + (
            other.n - other.p.get())),
        cap(n)
    {
        std::copy(
            other.p.get(),
            other.n,
            p.get());
    }
    vector(vector &&other)
        noexcept = default;
    void swap(vector &other) noexcept {
        std::swap(p, other.p);
        std::swap(n, other.n);
        std::swap(cap, other.cap);
    }
    vector& operator=(vector other) {
        other.swap(*this);
        return *this;
    }
    friend void swap(
        vector &left, vector &right)
        noexcept
    {
        left.swap(right);
    }
};
```

https://www.youtube.com/watch?v=7Qgd9B1KuMQ