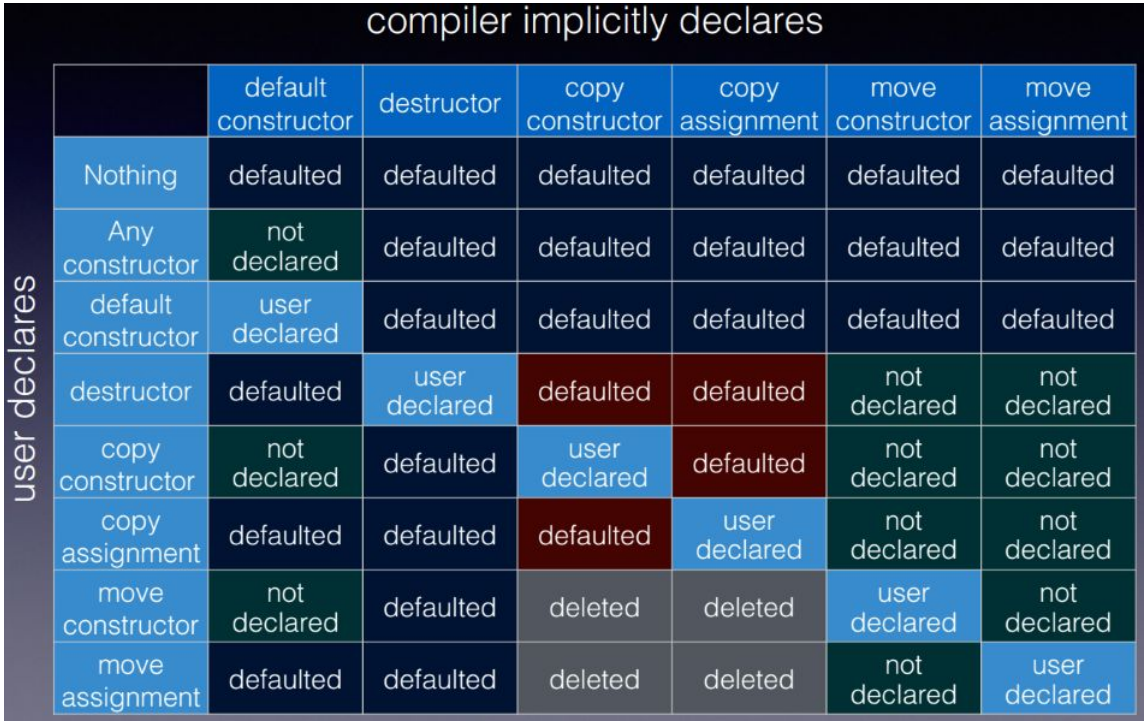
Объектно-ориентированное программирование

Object-oriented programming

IV. 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

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};
};
```

“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';
}
```

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

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;
    }
    // ...
};
```

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”

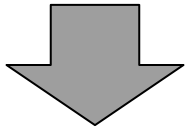
```
try {
    queue p;
    p.push(1);
    // throw here
} catch(const std::exception& e) {
    std::cerr << e.what() << '\n';
}
```

“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* (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);
```

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. */
```

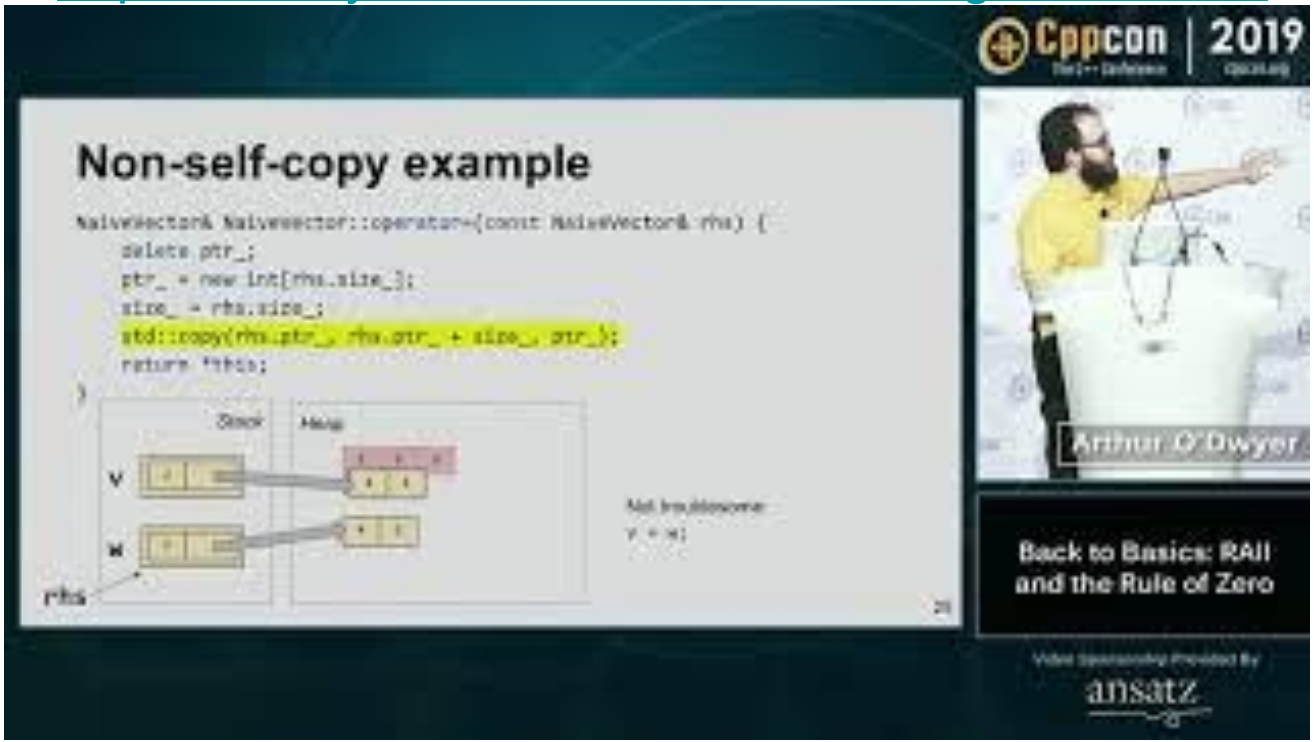
Правило “пяти” (rule of five)

```
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



V. Алгоритмы в C++

Algorithms

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

Итераторы

Iterators are a generalization of pointers that allow a C++ program to work with different data structures (for example, containers and ranges(since C++20)) in a uniform manner. The iterator library provides definitions for iterators, as well as iterator traits, adaptors, and utility functions.

Since iterators are an abstraction of pointers, their semantics are a generalization of most of the semantics of pointers in C++. This ensures that every function template that takes iterators works as well with regular pointers.

Iterator category	Operations and storage requirement						
	write	read	increment		decrement	random access	contiguous storage
			without multiple passes	with multiple passes			
LegacyIterator			Required				
LegacyOutputIterator	Required		Required				
LegacyInputIterator (mutable if supports write operation)		Required	Required				
LegacyForwardIterator (also satisfies *LegacyInputIterator*)		Required	Required	Required			
LegacyBidirectionalIterator (also satisfies *LegacyForwardIterator*)		Required	Required	Required	Required		
LegacyRandomAccessIterator (also satisfies *LegacyBidirectionalIterator*)		Required	Required	Required	Required	Required	
LegacyContiguousIterator[1] (also satisfies *LegacyRandomAccessIterator*)		Required	Required	Required	Required	Required	Required

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

Итераторы

```
template <class ForwardIterator, typename T>
void fill(ForwardIterator first, ForwardIterator last, const T& value) {
    for (; first != last; ++first) *first = value;
}
```

Итераторы

```
template <class ForwardIterator, typename T>
void fill(ForwardIterator first, ForwardIterator last, const T& value) {
    for (; first != last; ++first) *first = value;
}

class array {
public:
    struct iterator;
    iterator begin() { return iterator(data); }
    iterator end() { return iterator(data + 100); }
private:
    int data[100]{0};
};
```

Итераторы

```
template <class ForwardIterator, typename T>
void fill(ForwardIterator first, ForwardIterator last, const T& value) {
    for (; first != last; ++first) *first = value;
}
struct iterator {
private:
    int *p{nullptr};
public:
    iterator(int *ptr) : p(ptr) {}
    friend bool operator ==(iterator a, iterator b) { return a.p == b.p; }
    friend bool operator !=(iterator a, iterator b) { return !(a == b); }
    int &operator *() { return *p; }
    int *operator->();
    iterator &operator ++() { ++p; return *this; }
    iterator &operator ++(int);
};
```

Заголовочный файл <**algorithm**>

```
// std::for_each, std::for_each_n

vector v = {1, 2, 3, 4, 5, 6};

for_each(v.begin(), v.end(),
         [](const int& n) { cout << n << ' '; });

for_each_n(v.rbegin(), 4,
         [](const int& n) { cout << n << ' '; });
```

Заголовочный файл <**algorithm**>

```
// std::for_each, std::for_each_n

vector v = {1, 2, 3, 4, 5, 6};

for_each(v.begin(), v.end(),
         [](const int& n) { cout << n << ' '; });
// 1 2 3 4 5 6
for_each_n(v.rbegin(), 4,
         [](const int& n) { cout << n << ' '; });
// 6 5 4 3
```

Заголовочный файл <algorithm>

```
// std::all_of, std::any_of, std::none_of
struct is_odd {
    bool operator()(int x) const { return x % 2; }
};

vector v = {1, 2, 3, 4, 5, 6};

assert(none_of(v.cbegin(), v.cend(), is_odd()));
assert(all_of(v.cbegin(), v.cend(), is_odd()));
assert(any_of(v.cbegin(), v.cend(), is_odd()));
```

Заголовочный файл <algorithm>

```
// std::all_of, std::any_of, std::none_of
struct is_odd {
    bool operator()(int x) const { return x % 2; }
};

vector v = {1, 2, 3, 4, 5, 6};

assert(none_of(v.cbegin(), v.cend(), is_odd()));
assert(all_of(v.cbegin(), v.cend(), is_odd()));
assert(any_of(v.cbegin(), v.cend(), is_odd()));
```

Заголовочный файл <**algorithm**>

```
template<typename _InputIterator, typename _Predicate>
bool any_of(_InputIterator __first,
            _InputIterator __last,
            _Predicate __pred)
{ return !std::none_of(__first, __last, __pred); }

template<typename _InputIterator, typename _Predicate>
bool none_of(_InputIterator __first,
             _InputIterator __last,
             _Predicate __pred)
{ return __last == ::find_if(__first, __last, __pred); }
```

Заголовочный файл <algorithm>

```
// std::find, std::find_if, std::find_if_not
struct is_odd {
    bool operator()(int x) const { return x % 2; }
};

vector v = {1, 2, 3, 4, 5, 6};

find(v.begin(), v.end(), 3);
find_if(v.begin(), v.end(), is_odd());
find_if_not(v.begin(), v.end(), is_odd());
```

Заголовочный файл <algorithm>

```
// std::find, std::find_if, std::find_if_not
struct is_odd {
    bool operator()(int x) const { return x % 2; }
};

vector v = {1, 2, 3, 4, 5, 6};

find(v.begin(), v.end(), 3);                  // v.begin() + 2
find_if(v.begin(), v.end(), is_odd());        // v.begin() + 0
find_if_not(v.begin(), v.end(), is_odd()); // v.begin() + 1
```

Заголовочный файл <**algorithm**>

```
// std::count, std::count_if, std::equal, std::mismatch
vector v = {1, 2, 3, 4, 5, 6};

count(v.begin(), v.end(), 5);
count_if(v.begin(), v.end(), is_odd());

vector w = {1, 2, 3, 4, 4};

assert(equal(v.begin(), v.end() - 2, w.begin()));
mismatch(v.begin(), v.end(), w.begin());
```

Заголовочный файл <**algorithm**>

```
// std::count, std::count_if, std::equal, std::mismatch
vector v = {1, 2, 3, 4, 5, 6};

count(v.begin(), v.end(), 5);            // 1
count_if(v.begin(), v.end(), is_odd()); // 3

vector w = {1, 2, 3, 4, 4};

assert(equal(v.begin(), v.end() - 2, w.begin()));
mismatch(v.begin(), v.end(), w.begin()); // v.begin() + 4
                                         // w.begin() + 4
```

Заголовочный файл <**algorithm**>

```
// std::search, std::search_n

vector v = {1, 2, 3, 4, 5, 6};

search(v.begin(), v.end(), w.begin() + 1, w.end() - 2);

vector w = {1, 2, 3, 4, 4};
//      [first      last) count value
search_n(w.begin(), w.end(), 2, 4);
```

Заголовочный файл <**algorithm**>

```
// std::search, std::search_n

vector v = {1, 2, 3, 4, 5, 6};

search(v.begin(), v.end(), w.begin() + 1, w.end() - 2);
// v.begin() + 1
vector w = {1, 2, 3, 4, 4};

search_n(w.begin(), w.end(), 2, 4);
//  w.begin() + 3
```

Заголовочный файл <algorithm>

```
// std::copy, std::copy_if, std::copy_n, std::copy_backward
struct is_odd {
    bool operator()(int x) const { return x % 2; }
};

vector v = {1, 2, 3, 4, 5, 6}; vector w = {1, 2, 3, 4, 4};

copy(w.begin(), w.end(), v.begin());

copy_if(w.begin(), w.end(), v.begin(), is_odd());

copy_n(w.rbegin(), 3, v.begin());
```

Заголовочный файл <algorithm>

```
// std::copy, std::copy_if, std::copy_n, std::copy_backward
struct is_odd {
    bool operator()(int x) const { return x % 2; }
};

vector v = {1, 2, 3, 4, 5, 6}; vector w = {1, 2, 3, 4, 4};

copy(w.begin(), w.end(), v.begin());
// v[1, 2, 3, 4, 4, 6]
copy_if(w.begin(), w.end(), v.begin(), is_odd());
// v[1, 3, 3, 4, 4, 6]
copy_n(w.rbegin(), 3, v.begin());
// v[4, 4, 3, 4, 5, 6]
```

Заголовочный файл <**algorithm**>

```
// std::move, std::unique, std::remove, std::remove_if
// std::reverse, std::rotate
vector v = {1, 2, 3, 4, 5, 6}; vector w = {1, 2, 2, 3, 3};

move(w.begin(), w.end(), ::back_inserter(v));

unique(w.begin(), w.end());
remove(w.begin(), w.end(), 2);
remove_if(v.begin(), v.end(), is_odd());

reverse(v.begin(), v.end());
rotate(v.begin(), v.begin() + v.size() / 2, v.end());
```

Заголовочный файл <algorithm>

```
// std::move, std::unique, std::remove, std::remove_if
// std::reverse, std::rotate
vector v = {1, 2, 3, 4, 5, 6}; vector w = {1, 2, 2, 3, 3};

move(w.begin(), w.end(), ::back_inserter(v));
// v[1, 2, 3, 4, 5, 6, 1, 2, 3, 3]
unique(w.begin(), w.end()); // w[1, 2, 3, 3, 3]
remove(w.begin(), w.end(), 2); // w[1, 3, 3, 3, 3]
remove_if(v.begin(), v.end(), is_odd());
// v[2, 4, 6, 4, 5, 6]
reverse(v.begin(), v.end()); // v[6, 5, 4, 3, 2, 1]
rotate(v.begin(), v.begin() + v.size() / 2, v.end());
// v[4, 5, 6, 1, 2, 3]
```

Заголовочный файл <algorithm>

```
// std::transform, std::replace, std::replace_if
// std::fill, std::fill_n
struct is_odd {
    bool operator()(int x) const { return x % 2; }
};

int inc(int i) { return i + 1; }

vector v = {1, 2, 3, 4, 5, 6}; vector w = {1, 2, 2, 3, 3};

transform(w.begin(), w.end(), v.begin(), inc);
replace_if(v.begin(), v.end(), is_odd(), 0);
fill(w.begin(), w.end(), 0);
```

Заголовочный файл <**algorithm**>

```
// std::transform, std::replace, std::replace_if
// std::fill, std::fill_n
vector v = {1, 2, 3, 4, 5, 6}; vector w = {1, 2, 2, 3, 3};

transform(w.begin(), w.end(), v.begin(), inc);
// v[2, 3, 3, 4, 4, 6]
replace_if(v.begin(), v.end(), is_odd(), 0);
// v[0, 2, 0, 4, 0, 6]
fill(w.begin(), w.end(), 0);
// w[0, 0, 0, 0, 0]
```

Заголовочный файл <algorithm>

```
// std::shuffle, std::generate, std::generate_n
#include <random> // std::mt19937, std::random_device
vector v = {1, 2, 3, 4, 5, 6}; vector w = {1, 2, 3, 4, 4};

random_device rd;
mt19937 g(rd());

shuffle(v.begin(), v.end(), g);
generate(v.begin(), v.end(), random);
generate_n(v.begin(), 3, random);
```