Move-семантика

Правило пяти. Правило нуля.

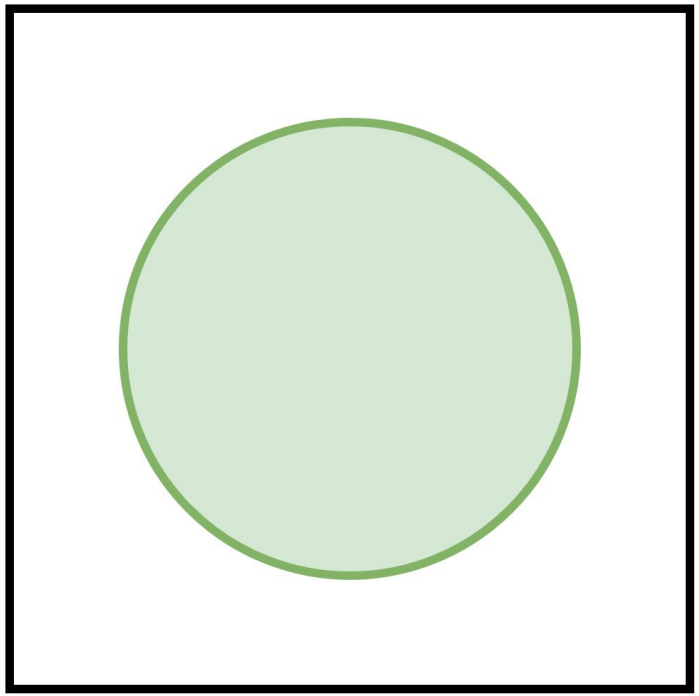
На прошлой лекции

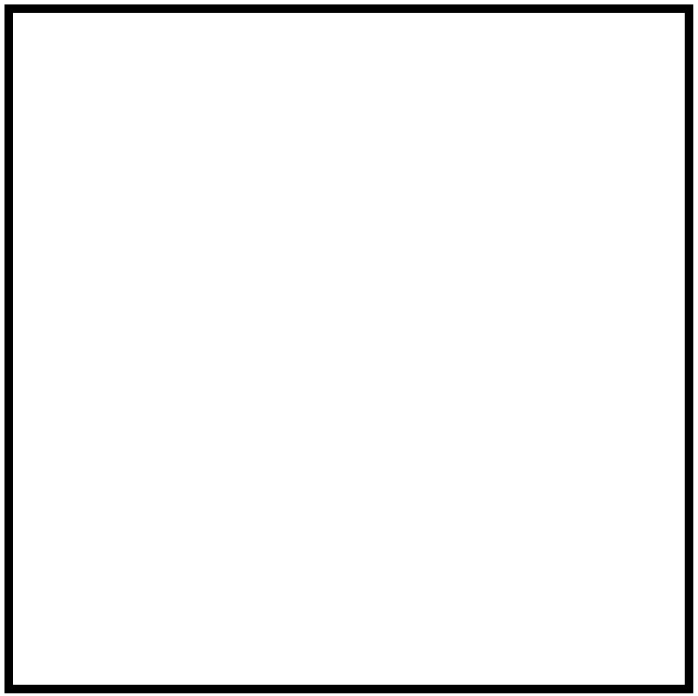
```
struct Widget {
  Widget() = default;

  ~Widget() = default;
  Widget(const Widget&) = default;
  Widget(Widget&&) = default;
  Widget& operator=(const Widget&) = default;
  Widget& operator=(Widget&&) = default;
};
```

Разобрали правило трех

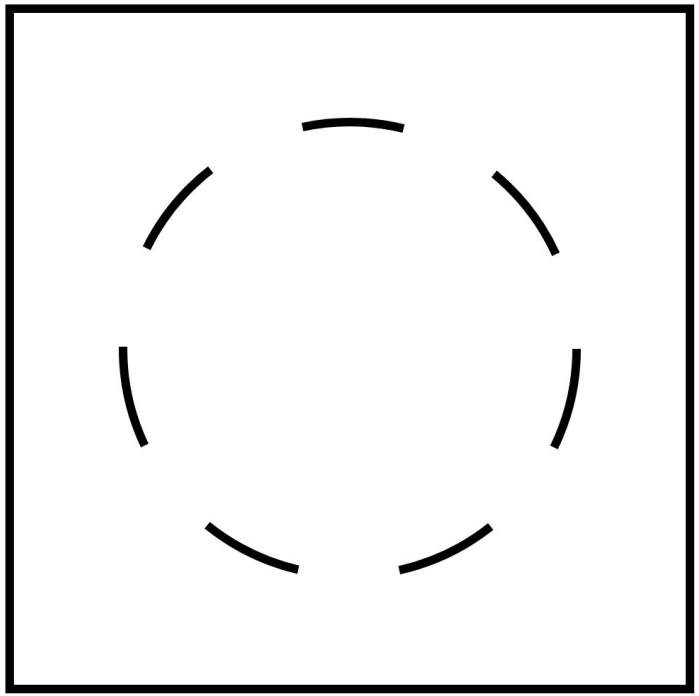
Сегодня – правило пяти

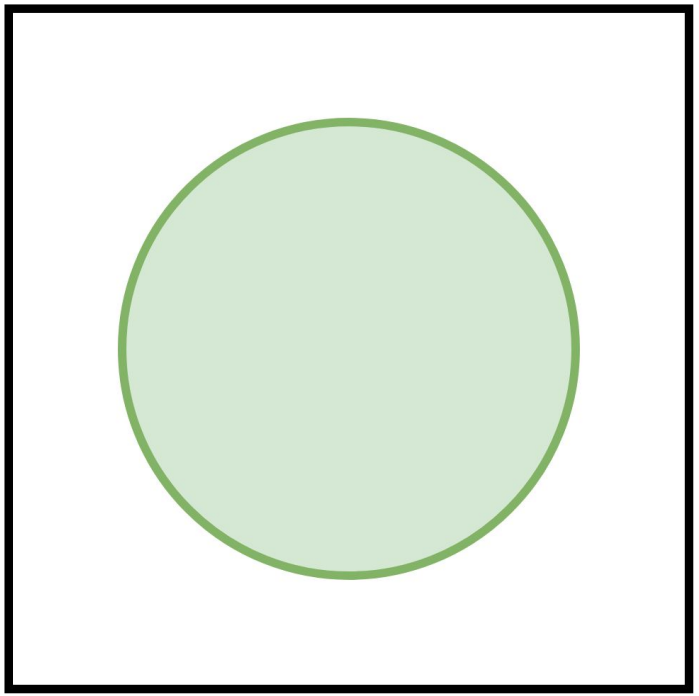
Move-семантика для 4-летних



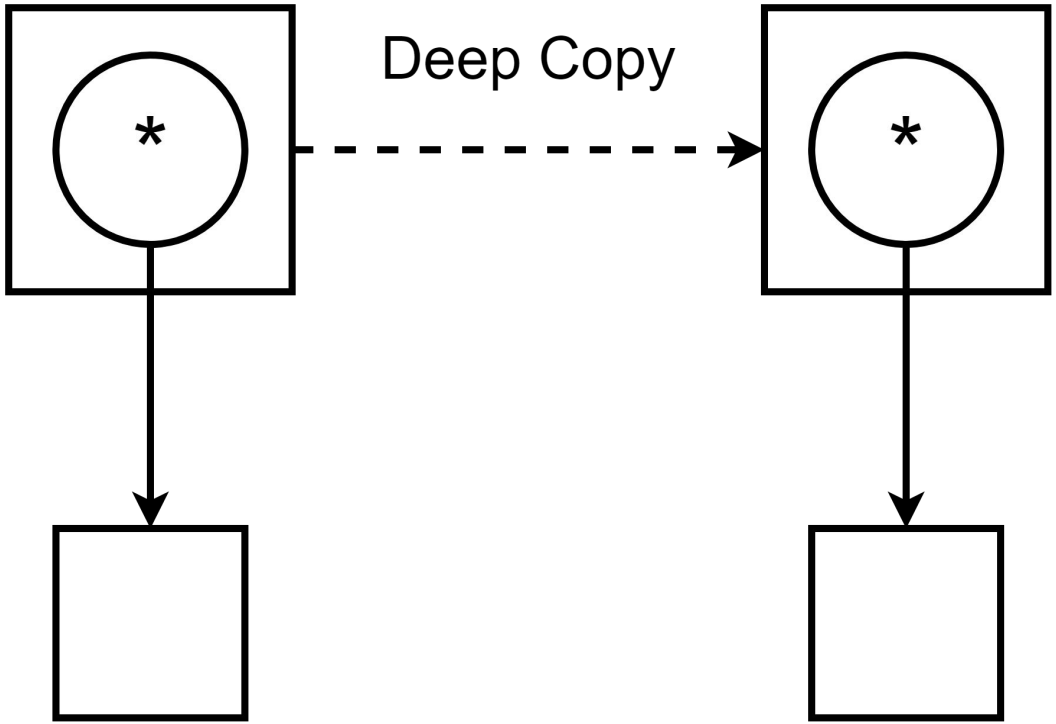


Move-семантика для 4-летних

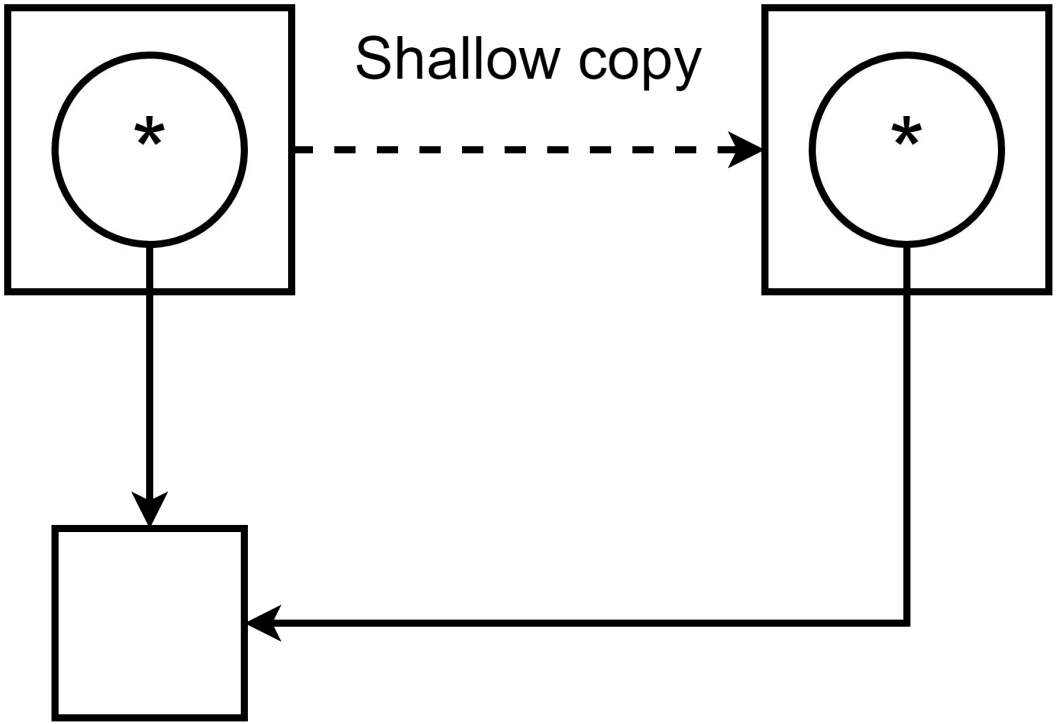




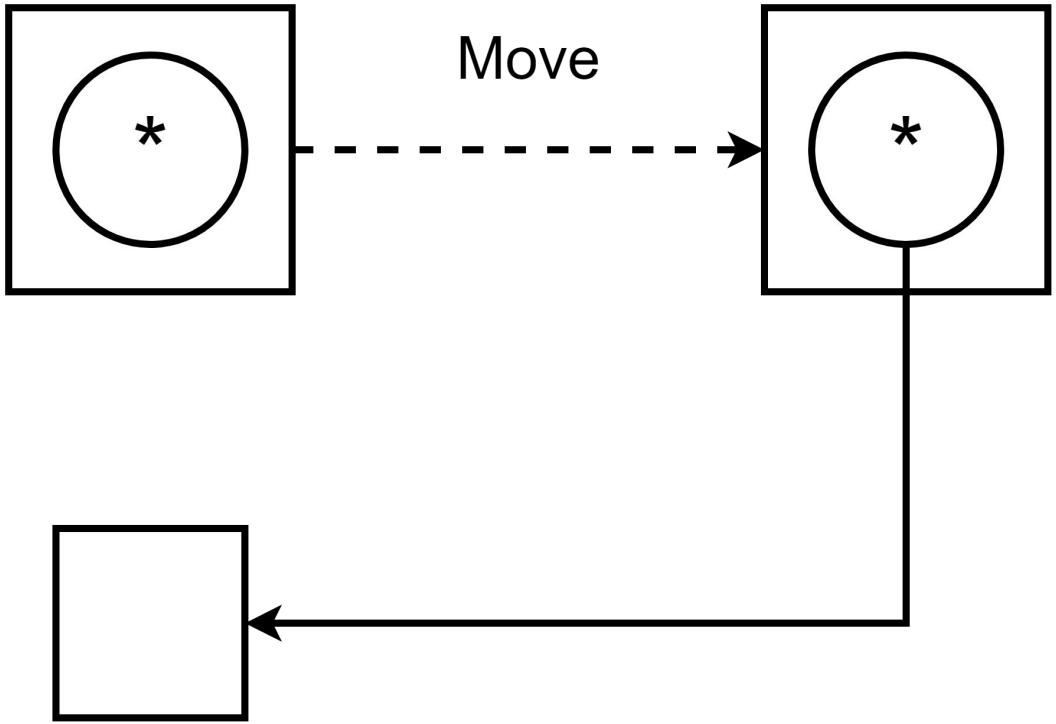
Копирование для 4-летних C++-разработчиков



Копирование для 4-летних C++-разработчиков



Перемещение для 4-летних C++-разработчиков



Когда полезно перемещение

```
class MyStr {
public:
  ~MyStr();
  MyStr(const MyStr&);
  MyStr& operator=(const MyStr&);
};
```

```
MyStr get_hostname();

MyStr s = "hello";
s = get_hostname();
```

Что будет вызвано в этой строке?

Когда полезно перемещение

```
class MyStr {
public:
  ~MyStr();
  MyStr(const MyStr&);
  MyStr& operator=(const MyStr&);
};
```

```
MyStr get_hostname();

MyStr s = "hello";
s = get_hostname();
```

Что будет вызвано в этой строке?

Когда полезно перемещение

```
class MyStr {
public:
  ~MyStr();
  MyStr(const MyStr&);
  MyStr& operator=(const MyStr&);
};
```

```
MyStr get_hostname();

MyStr s = "hello";
s = get_hostname();
```

Copy assignment

Когда полезно перемещение

```
class MyStr {
public:
  ~MyStr();
  MyStr(const MyStr&);
  MyStr& operator=(const MyStr&);
};
```

```
MyStr get_hostname();

MyStr s = "hello";
s = get_hostname();
```

1. Создание временного объекта
2. Копирование из временного объекта в s
3. Уничтожение временного объекта

Когда полезно перемещение

```
class MyStr {
public:
  ~MyStr();
  MyStr(const MyStr&);
  MyStr& operator=(const MyStr&);
};
```

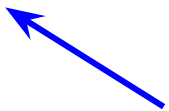
```
MyStr get_hostname();

MyStr s = "hello";
s = get_hostname();
```

Нам нужно отличать ситуацию, когда
справа от присваивания — временный объект

Когда полезно перемещение

```
class MyStr {
public:
  ~MyStr();
  MyStr(const MyStr&);
  MyStr& operator=(const MyStr&);
  MyStr& operator=(MyStr&&);
};
```

```
MyStr get_hostname();

MyStr s = "hello";
s = get_hostname();
```



rvalue ссылки связываются в том числе с временными объектами

Выражения

Выражение (expression) — последовательность операторов и операндов, определяющих процесс вычислений.

Характеристики выражений:

- Тип данных
- Категория выражения (value category)

Примеры выражений

```
// int x, a = 40, b = 2;
// float z = 2.0f;

a + b       // Возвращает значение типа int
a + z       // float
x = a + b   // возвращает x
a
42
f()
v[42]
```

Намеренно не ставим точки с запятой, чтобы не получить statement

История: язык C

Термин lvalue происходит из выражения присваивания:

```
E1 = E2
```

Левый операнд должен быть lvalue,
правый — rvalue.

Примеры в языке C

```
x = 42      // Ok
x = y       // Ok: lvalue to rvalue conversion
42 = x      // Error: lvalue required as left operand
a + b = x   // Error: ...
f() = x     // Error: ... Но это может работать в C++
```

Предварительный вывод:

- lvalue связан с объектом
- rvalue — со значением

C++ усложняет правила

```
const int x = 42;

x = 4;  // x — lvalue, но константы нельзя менять


int g = ...;

int& f() { return g; }

f() = 42;  // Ok
```

Это похоже на std::vector::operator[]

lvalue, rvalue

Это эвристика, а не строгий критерий

lvalue:

- Можно получить адрес
- Есть имя, выбранное разработчиком

rvalue:

- Нельзя получить адрес
  - `&(a+b), &42, &this`
- Имя — ключевое слово
  - `&this`

lvalue-ссылки

```
int x = 42;

int& rx = x;
const int& crx = x;
int& rrx = rx;

const int& crv = 42;

int& rv = 42;
```

lvalue-ссылки

```
int x = 42;

int& rx = x;            // Ok
const int& crx = x;
int& rrx = rx;

const int& crv = 42;

int& rv = 42;
```

lvalue-ссылки

```
int x = 42;

int& rx = x;           // Ok
const int& crx = x;    // Ok
int& rrx = rx;

const int& crv = 42;

int& rv = 42;
```

lvalue-ссылки

```
int x = 42;

int& rx = x;           // Ok
const int& crx = x;    // Ok
int& rrx = rx;         // Ok

const int& crv = 42;

int& rv = 42;
```

lvalue-ссылки

```
int x = 42;

int& rx = x;           // Ok
const int& crx = x;    // Ok
int& rrx = rx;         // Ok

const int& crv = 42;   // Ok, exception!

int& rv = 42;
```

lvalue-ссылки

```
int x = 42;

int& rx = x;            // Ok
const int& crx = x;     // Ok
int& rrx = rx;          // Ok

const int& crv = 42;    // Ok, exception!

int& rv = 42;           // Error: cannot bind
                        // non-const lvalue reference
                        // to an rvalue
```

rvalue-ссылки

```
int x = 2;

int&& rx = 42;

int&& y = x + 1;

const int&& q = 42;

int&& z = y;

int&& z = std::move(y);
```

rvalue-ссылки

```
int x = 2;

int&& rx = 42;              // Ok

int&& y = x + 1;

const int&& q = 42;

int&& z = y;

int&& z = std::move(y);
```

rvalue-ссылки

```
int x = 2;

int&& rx = 42;              // Ok

int&& y = x + 1;           // Ok

const int&& q = 42;

int&& z = y;

int&& z = std::move(y);
```

rvalue-ссылки

```
int x = 2;

int&& rx = 42;              // Ok

int&& y = x + 1;           // Ok

const int&& q = 42;        // Ok, but useless

int&& z = y;

int&& z = std::move(y);
```

rvalue-ссылки

```
int x = 2;

int&& rx = 42;              // Ok

int&& y = x + 1;           // Ok

const int&& q = 42;        // Ok, but useless

int&& z = y;               // Error

int&& z = std::move(y);
```

rvalue-ссылки

```
int x = 2;

int&& rx = 42;              // Ok

int&& y = x + 1;           // Ok

const int&& q = 42;        // Ok, but useless

int&& z = y;               // Error

int&& z = std::move(y); // Ok, but
                           // std::move does not move
```

rvalue to const lvalue-ref

```
void f(std::string s);          // s – входной параметр

std::string s1 = “...”, s2 = “...”;

f(s1);        // Ok, call with lvalue
f(s1 + s2);   // Ok, call with rvalue
```

rvalue to const lvalue-ref

```
void f(const std::string& s); // s – входной параметр

std::string s1 = “...”, s2 = “...”;

f(s1);       // Ok, call with lvalue
f(s1 + s2);  // Still Ok, call with rvalue
```

Смысл параметра не изменился,
клиентский код должен продолжать работать.

Связывание ссылки с временным объектом

При таком связывании происходит материализация временного объекта:

```
const int& x = 42;
int&& y = 42;

// Можем получить адреса &x, &y
```

Выбор перегрузки

```
void f(int&);
void f(int&&);

int main() {
  int x = 42;
  int& rx = x;
  int&& rrx = 42;

  f(x);                    // ?
  f(rx);                   // ?
  f(rrx);                  // ?
}
```

Выбор перегрузки

```
void f(int&);
void f(int&&);

int main() {
  int x = 42;
  int& rx = x;
  int&& rrx = 42;

  f(x);                    // f(int&)
  f(rx);                   // ?
  f(rrx);                  // ?
}
```

Выбор перегрузки

```
void f(int&);
void f(int&&);

int main() {
  int x = 42;
  int& rx = x;
  int&& rrx = 42;

  f(x);                    // f(int&)
  f(rx);                   // f(int&)
  f(rrx);                  // ?
}
```

Выбор перегрузки

```
void f(int&);
void f(int&&);

int main() {
  int x = 42;
  int& rx = x;
  int&& rrx = 42;

  f(x);
  f(rx);
  f(rrx);
}
```

Почему так?

Как вызвать f(int&&)?

```
// f(int&)
// f(int&)
// f(int&)
```


https://godbolt.org/z/n9qz1rKrj

Выбор перегрузки

```
void f(int&);
void f(int&&);

int main() {
  int x = 42;
  int& rx = x;
  int&& rrx = 42;

  f(x);                              // f(int&)
  f(rx);                             // f(int&)
  f(static_cast<int&&>(rrx));        // f(int&&)
}
```

Выбор перегрузки

```
void f(int&);
void f(int&&);

int main() {
  int x = 42;
  int& rx = x;
  int&& rrx = 42;

  f(x);                    // f(int&)
  f(rx);                   // f(int&)
  f(std::move(rrx));       // f(int&&)
}
```

std::move does not move

```
template<typename _Tp>
[[nodiscard] constexpr
typename std::remove_reference<_Tp>::type&&
move(_Tp&& __t) noexcept {
  return static_cast<
      typename std::remove_reference<_Tp>::type&&
  >(__t);
}
```


GCC: move.h#L104

Последнее замечание

auto&& – это не rvalue-ссылка

Действует другой набор правил

Move constructor

Move-конструктор оставляет свой аргумент в консистентном, но неопределенном состоянии.

```
MyString(MyString&& other)
  : length_(other.length_),
    str_(other.str_) {
  other.length_ = 0;
  other.str_ = nullptr;
}
```

Move assignment

```
MyString& operator=(MyString&& other) {
  if (this == &other) return *this;

  delete[] str_;
  str_ = other.str_;
  length_ = other.length_;

  other.length_ = 0;
  other.str_ = nullptr;
  return *this;
}
```

Move assignment

```
MyString& operator=(MyString&& other) {
  if (this == &other) return *this;

  delete[] str_;
  str_ = std::exchange(other.str_, nullptr);
  length_ = std::exchange(other.length_, 0);

  return *this;
}
```


https://godbolt.org/z/WYTdf8s4T

Flashforward

Приведенные специальные методы можно написать изящнее.

Мы вернемся к этому весной.

Генерация специальных методов компилятором

[Becoming a Rule of Zero Hero — Glennan Carnie]

Rule of Five

Если в классе определен или удален любой из методов:

- Деструктор
- Копирующий конструктор
- Перемещающий конструктор
- Копирующий оператор присваивания
- Перемещающий оператор присваивания

то следует определить или удалить все пять.

CppCoreGuidelines: C.21

Rule of Zero

1. Если вы можете обойтись без определения специальных методов, то не определяйте их.
2. Если в классе определены специальные методы, то по Single Responsibility Principle в нем не должно быть никаких других методов.

CppCoreGuidelines: C.20

Пример из CppCoreGuidelines

```
struct Named_map {
 public:
  // ... no default operations declared …
 private:
  std::string name;
  std::map<int, int> rep;
};

Named_map nm;         // default construct
Named_map nm2 {nm};   // copy construct
```

copy-swap (C++03)

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

Сколько здесь копирований?

https://godbolt.org/z/M5M5679s7

std::swap (C++11)

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

Сколько здесь копирований?

https://godbolt.org/z/jKTPxsxq9

Передача параметров

```
struct Widget {
  Widget(const Tracer& t)
    : t_(t) {}
  Tracer t_;
};

Widget w1(t); // lvalue
```

1. lvalue связывается со ссылкой.
2. Tracer(const Tracer&)

Передача параметров

```
struct Widget {
  Widget(const Tracer& t)
    : t_(t) {}
  Tracer t_;
};
```

```
Tracer(int)
Tracer(const Tracer&)
~Tracer()
```

```
Widget w1(42); // rvalue
```

Передача параметров

```
struct Widget {
  Widget(Tracer t)
    : t_(std::move(t)) {}
  Tracer t_;
};
```

```
Widget w1(t); // lvalue
```

```
Tracer(const Tracer&)
Tracer(Tracer&&)
~Tracer()
```

Можно считать, что стало не хуже (move – легковесная операция)

Передача параметров

```
struct Widget {
  Widget(Tracer t)
    : t_(std::move(t)) {}
  Tracer t_;
};
```

```
Widget w1(42); // rvalue
```

```
Tracer::Tracer(int)
Tracer::Tracer(Tracer&&)
Tracer::~Tracer()
```

Стало лучше
(избавились от копирования)

Передача параметров

```
struct Widget {
  Widget(const Tracer& t)
   : t_(t) {}

  Widget(Tracer&& t)
   : t_(std::move(t)) {}

  Tracer t_;
};
```

Лучшее от двух решений

Но дублирование кода

RVO

```
Tracer f() {
  Tracer t;
  return t;
}

int main() {
  Tracer t = f();
}
```

Что вызовется?

RVO

```
Tracer f() {
  Tracer t;
  return t;
}

int main() {
  Tracer t = f();
}
```

```
Tracer::Tracer()
Tracer::~Tracer()
```


https://godbolt.org/z/vK5x3hanx

RVO

```
Tracer f() {
  Tracer t;
  return std::move(t);
}

int main() {
  Tracer t = f();
}
```

```
Tracer::Tracer()
Tracer::Tracer(Tracer&&)
Tracer::~Tracer()
Tracer::~Tracer()
```

Не «помогайте» компилятору

https://godbolt.org/z/Kbqe8hE7T

RAII

Resource Acquisition Is Initialization

1. Получение ресурса — инициализация объекта
2. Освобождение ресурса — разрушение объекта

Примеры:

```
std::string, std::map, std::vector, …

std::ifstream, std::ofstream, …

std::mutex m;
std::lock_guard lock(m);
```

Некопируемый ресурс

```
class Socket {
 public:
  ~Socket();

  Socket(const Socket&) = delete;
  Socket& operator=(const Socket&) = delete;

  Socket(Socket&& other);
  Socket& operator=(Socket&& other);
};
```

Q&A