Инициализация и копирование

rvalue-ссылки. Move-семантика. Правило 5 (0).

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

lvalue:

- Можно получить адрес

rvalue:

- Нельзя получить адрес
  - `&(a+b), &42, &this`

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&& x = 42;                // Ok

int&& y = a + 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(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&&)
}
```


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


https://godbolt.org/z/n9qz1rKrj

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);
}
```

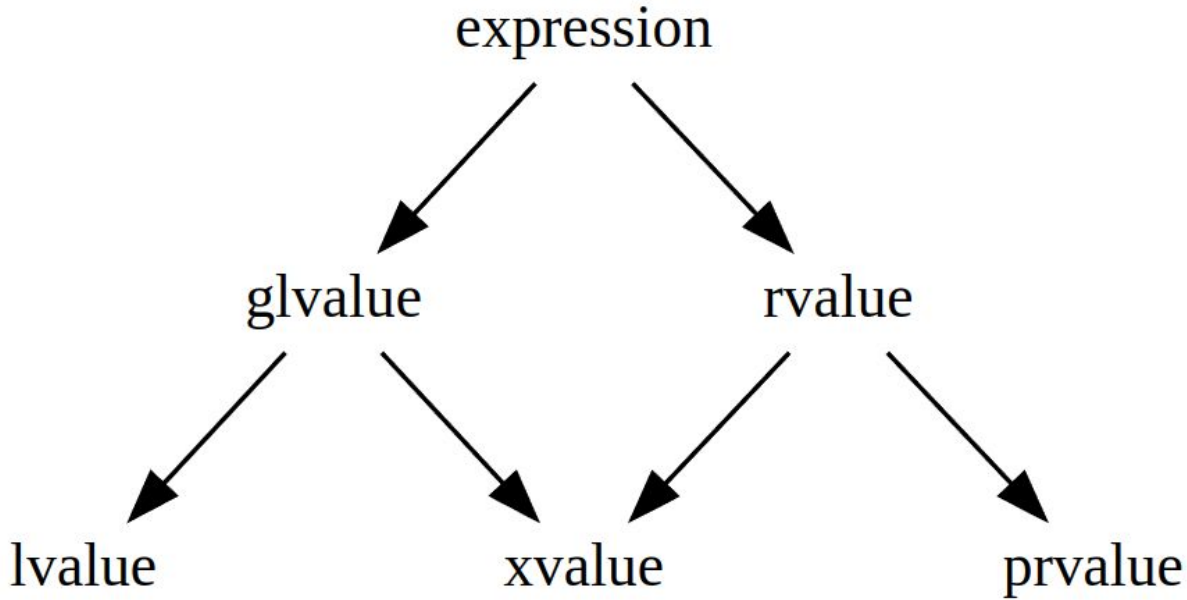

GCC: move.h#L104

Последнее замечание

auto&& – это не rvalue-ссылка

Действует другой набор правил

Value category [basic.lval]



[basic.lval]

Покритикуйте этот код

```
class MyString {
 public:
  MyString(const char* str)
      : length_(std::strlen(str)),
        str_(new char[length_ + 1]()) {
    std::copy(str, str + length_, str_);
  }
 private:
  ...
};
```

Покритикуйте этот код

```
class MyString {
 public:
  MyString(const char* str)
      : length_(std::strlen(str)),
        str_(new char[length_ + 1]()) {
    std::copy(str, str + length_, str_);
  }
 private:
  ...
};
```

Нет парного освобождения памяти

https://godbolt.org/z/3x6x4xorf

Деструктор

```
class MyString {
 public:

  ~MyString() {
    delete[] str_;
  }

 private:
  ...
};
```

Еще проблемы?

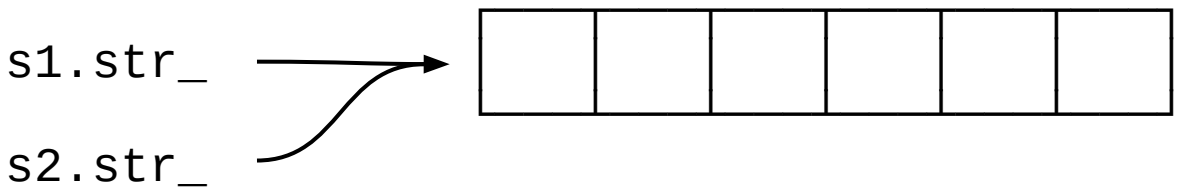
Что происходит в памяти?

```
MyString s1 = "Hello, World";

MyString s2(s1);
```

Что происходит в памяти?

```
MyString s1 = "Hello, World";
MyString s2(s1);
```



Конструктор копирования

```
class MyString {
 public:
  MyString(const MyString& other)
    : length_(other.length_),
      str_(new char[length_ + 1]()) {
    std::copy(other.str_, other.str_ + length_, str_);
  }
};
```

Устраняем дублирование

```
class MyString {
 public:
  MyString(const MyString& other)
     : MyString(other.str_) { }
};
```


https://godbolt.org/z/rK7hG8scx

Что происходит в памяти?

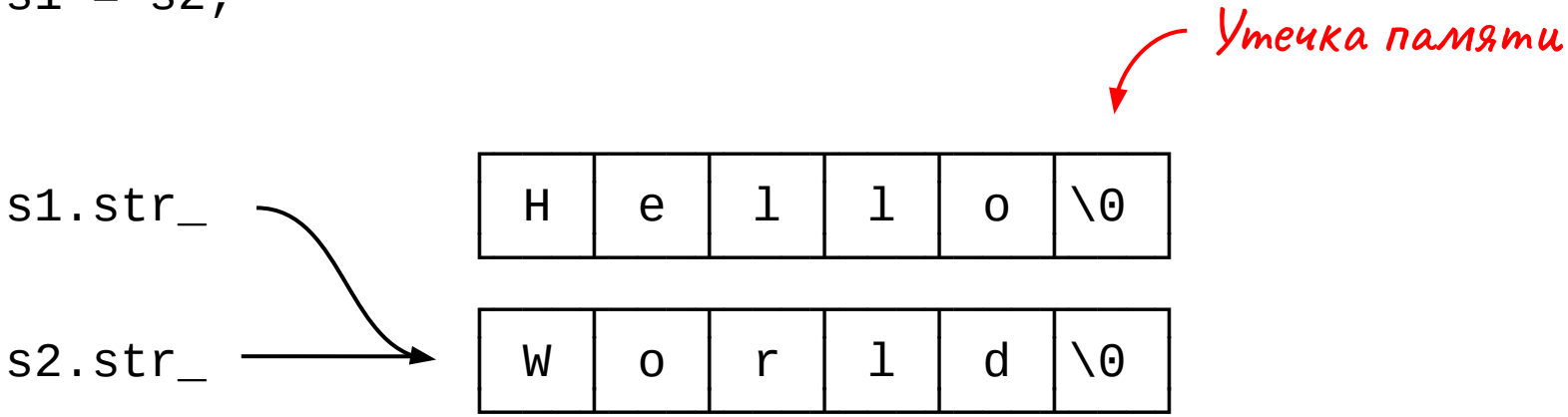
```
MyString s1 = "Hello”, s2 = “World";

s1 = s2;
```

Что происходит в памяти?

```
MyString s1 = "Hello”, s2 = “World";

s1 = s2;
```



https://godbolt.org/z/3YWdYeoa3

Copy Assignment

```
MyString& operator=(const MyString& other) {
  delete[] str_;
  str_ = new char[other.length_ + 1]();
  length_ = other.length_;
  std::copy(other.str_, other.str_ + length_, str_);
  str_[length_] = '\0';
  return *this;
}
```


https://godbolt.org/z/jnrYWhfWh

Copy Assignment

```
MyString& operator=(const MyString& other) {
  delete[] str_;
  str_ = new char[other.length_ + 1]();
  length_ = other.length_;
  std::copy(other.str_, other.str_ + length_, str_);
  str_[length_] = '\0';
  return *this;
}
```

Можем оптимизировать аллокации?

https://godbolt.org/z/jnrYWhfWh

Copy Assignment

```
MyString& operator=(const MyString& other) {
  if (length_ < other.length_) {
    delete[] str_;
    str_ = new char[other.length_ + 1]();
  }
  length_ = other.length_;
  std::copy(other.str_, other.str_ +length_, str_);
  str_[length_] = '\0';
  return *this;
}
```

TODO: capacity_

Copy Assignment

```
MyString& operator=(const MyString& other) {
  delete[] str_;
  str_ = new char[other.length_ + 1]();
  length_ = other.length_;
  std::copy(other.str_, other.str_ + length_, str_);
  str_[length_] = '\0';
  return *this;
}
```

Какие еще проблемы?

https://godbolt.org/z/jnrYWhfWh

Self Assignment

```
MyString s1 = "Hello";
s1 = s1;
std::cout << s1 << '\n';
```

Упражнение читателю: нарисуйте схему памяти

https://godbolt.org/z/nWYK7enb6

Self assignment

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


https://godbolt.org/z/19s3xoeME

Copy Assignment

- Реализованное присваивание все еще недостаточно хорошо
- Другой подход мы разберем в теме «Исключения»

C++03: Rule Of Three

Если в классе реализован хотя бы один из специальных методов:

- Деструктор
- Конструктор копирования
- Копирующий оператор присваивания

то следует реализовать все три.

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

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;
}
```

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);
}
```

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

Передача параметров в функцию

	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD**[]**, array<BigPOD>)
Out	X f()		
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain "copy"			

"Cheap" ≈ a handful of hot int copies
"Moderate cost" ≈ memcpy hot/contiguous ~1KB and no allocation

** or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

CppCoreGuidelines

Передача параметров в функцию

<table>
<tr><th></th><th>Cheap or impossible to copy (e.g., int, unique_ptr)</th><th>Cheap to move (e.g., vector<T>, string)
or Moderate cost to move (e.g., array<vector>, BigPOD)
or Don't know (e.g., unfamiliar type, template)</th><th>Expensive to move (e.g., BigPOD[], array<BigPOD>)</th></tr>
<tr><td>Out</td><td colspan="2">X f()</td><td></td></tr>
<tr><td>In/Out</td><td colspan="3">f(X&)</td></tr>
<tr><td>In</td><td rowspan="2">f(X)</td><td colspan="2">f(const X&)</td></tr>
<tr><td>In & retain copy</td><td colspan="2">f(const X&) + f(X&&) & move **</td></tr>
<tr><td>In & move from</td><td colspan="3">f(X&&) **</td></tr>
</table>

** or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

*** special cases can also use perfect forwarding (e.g., multiple in+copy params, conversions)*

CppCoreGuidelines

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);
};
```

Что будет выведено?

```
struct Widget {};

struct Window {
  Window(Widget w1, Widget w2) {}
  int x_ = 0;
};

int main() {
  Window w(Widget(), Widget());
  std::cout << w.x_ << '\n';
}
```

Most vexing parse

```
struct Widget {};

struct Window {
  Window(Widget w1, Widget w2) {}
  int x_ = 0;
};

int main() {
  Window w(Widget(), Widget());
  std::cout << w.x_ << '\n';
}
```

Ошибка компиляции

https://godbolt.org/z/MPYoh1jc7

Most vexing parse

```
struct Widget {};

struct Window {
  Window(Widget w1, Widget w2) {}
  int x_ = 0;
};

int main() {
  Window w{Widget(), Widget()};
  std::cout << w.x_ << '\n';
}
```

Проблема: инициализация контейнеров

```
std::vector<int> v;
v.push_back(3);
v.push_back(14);
v.push_back(15);
v.push_back(92);
v.push_back(65);

// Но...
int pi[] = {3, 14, 15, 92, 65};
```

Слишком громоздко

std::initializer_list

```
std::vector(std::initializer_list<T> init, …)
```

У такого конструктора приоритет при {}-инициализации.

В чем разница?

```
std::vector v1(3, 14);
std::vector v2{3, 14};
```

std::initializer_list

```
std::vector(std::initializer_list<T> init, …)
```

У такого конструктора приоритет при {}-инициализации.

В чем разница?

```
std::vector v1(3, 14);  // [14, 14, 14]
std::vector v2{3, 14};  // [3, 14]
```

Материалы

- CopperSpice: Modern C++ (value categories) https://www.youtube.com/watch?v=wkWtRDrjEH4
- Back to Basics: Understanding Value Categories - Ben Saks - CppCon 2019 https://www.youtube.com/watch?v=XS2JddPq7GQ