ЛЕКЦИЯ 13
КОНСТРУКТОРЫ И ДЕСТРУКТОРЫ

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

ЛЕКТОР ФУРМАВНИН С.А.

SLEEPING TEST

```
struct S {
private:
    int foo(int x) {return 1;}
public:
    int foo(float x) {return 2;}
};

int main() {
    S s;
    std::cout << s.foo(1) << std::endl;
    return 0;
}
```

SLEEPING TEST

```
class C {
private:
    class Inner {
    public:
        int x = 42;
    };
public:
    Inner foo() {return Inner();}
};

int main() {
    S s;
    std::cout << c.foo().x << std::endl;
    return 0;
}
```

СОЗДАНИЕ ОБЪЕКТА

```
struct vec2D {
public:
    int x, y;
};

vec2D s = {1, 2}; // ok v(1,2)
vec2D s = {1}; // ok v(1,0)
vec2D s {1}; // ok v(1,0) новшество C++11
```

СОЗДАНИЕ ОБЪЕКТА

```
struct vec2D {
private:
    int x, y;
};
```

Агрегация ломается при появлении приватного состояния

```
vec2D s = {1, 2}; // ошибка, это не агрегат
vec2D s = {1}; // ошибка, это не агрегат
vec2D s {1}; // ошибка, это не агрегат
```

Кроме того, теперь у нас нет уверенности, что поле x проинициализировано

СОЗДАНИЕ ОБЪЕКТА

```
struct vec2D {
public:
    int x;
private:
    int y;
};

vec2D s = {1, 2}; // ошибка, это не агрегат
vec2D s = {1}; // ошибка, это не агрегат
vec2D s {1}; // ошибка, это не агрегат
```

СОЗДАНИЕ ОБЪЕКТА

```
struct vec2D {
public:
    int x, y = 0;
    vec2D(int num) {x = num;} // конструктор
};
```

Он может быть инициализирован либо **direct** либо **copy** инициализацией

```
vec2D s (num); // прямая инициализация, старый синтакс
vec2D s {num}; // прямая инициализация, новый синтакс
vec2D s = 1; // копирующая инициализация
```

О {}

Фигурные скобки контекстно-зависимы.

1. Если ваш тип – агрегат, то {} – агрегатная инициализация
2. Если у вашего типа есть конструктор от списка инициализации, то {} - конструктор от списка от инициализации
3. Если у вашего типа есть какой-то конструктор, то {} – это какой-то конструктор

o {}

ПОНЯТИЕ КОНСТРУКТОРА

Конструктор – это компонентная функция, позволяющая устанавливать значения при инициализации объекта без необходимости обращаться к какой-либо функции члену.

Конструктор – это специальная функция-член, которая вызывается автоматически всякий раз, когда объект создается.

ОСОБЕННОСТИ КОНСТРУКТОРА

- Конструктор может быть только функцией-членом
- Конструктор не имеет возвращаемого типа значения
- Имя конструктора строго совпадает с именем класса (структуры)

СТАРАЯ ИНИЦИАЛИЗАЦИЯ

До 2011 года вызов конструктора предполагал круглые скобки

```
Triangle2D t(p1, p2, p3) // вызов конструктора
Triangle2D t{p1, p2, p3} // вызов конструктора
```

До сих пор это означает одно и то же. Но есть одно но!

```
myclass_t m(list_t(), list_t()); // вызов конструктора?
myclass_t m{list_t(), list_t()}; // вызов конструктора?
```

Одна из этих строчек значит не то, что вы думаете

ДВА ПРАВИЛА

- Все, что может быт засчитано, как функция, считается как функция
- Все, что может быть засчитано как обращение к полю, будет засчитано как обращение к полю

ДВОЙНАЯ ИНИЦИАЛИЗАЦИЯ

- Присваивая в теле конструктора, мы инициализируем дважды(второй раз временный объект для присваивания)

```
struct S{
    S() {std::cout << "default" << std::endl;}
    S(KeyT key) {std::cout << "direct" << std::endl;}
};

struct Node {
    S key_; int val_;
    Node(KeyT key, int val) {key_ = key; val_ = val;}
};
```

СПИСКИ ИНИЦИАЛИЗАЦИИ

- Чтобы уйти от двойной инициализации, до тела конструктора предусмотрены списки инициализации

```
struct S{
    S() {std::cout << "default" << std::endl;}
    S(KeyT key) {std::cout << "direct" << std::endl;}
};

struct Node {
    S key_; int val_;
    Node(KeyT key, int val) : key_(key), val_(val){}
};
```

ДВА ПРАВИЛА ИНИЦИАЛИЗАЦИИ

- Список инициализации выполняется строго в том порядке, в каком поля определены в классе (не в том, в каком они записаны в списке)

```
struct Node {
    S key_; T key2_;
    Node(KeyT key) : key2_(key), key_(key){} // S, T
};
```

ДВА ПРАВИЛА ИНИЦИАЛИЗАЦИИ

- Инициализация в теле класса незримо входит в список инициализации

```
struct Node {
    S key_ = 1; T key2_;
    Node(KeyT key) : key2_(key) {} // S, T
};
```

ПАРАМЕТРЫ ПО УМОЛЧАНИЮ

- Если уже что-то есть в списке инициализации, то инициализатор в теле класса игнорируется

```
struct Node {
    S key_ = 1;
    Node() {} // key_(1)
    Node(KeyT key) : key_(key) {} // key_(key)
};
```

ПАРАМЕТРЫ ПО УМОЛЧАНИЮ

- Такое лучше переписать с **параметром по умолчанию**

```
struct Node {
    S key_;
    Node(KeyT key = 1) : key_(key) {} // key_(key)
};
```

ДЕЛЕГАЦИЯ КОНСТРУКТОРОВ

- Если конструктор делает нетривиальные вещи, его можно **делегировать**

```
struct S {
    int max = 0, min = 0;
    S(int my_max) : max(my_max > 0 ? my_max : DEFAULT_MAX)
{}
    S(int my_max, int my_min) : S(my_max), min(my_max > 0 &&
my_min < max ? my_min : DEFAULT_MIN) {}
};
```

ДЕЛЕГАЦИЯ КОНСТРУКТОРОВ

```
struct S {
    int max = 0, min = 0;
    S(int my_max) : max(my_max > 0 ? my_max : DEFAULT_MAX)
{}
    S(int my_max, int my_min) : S(my_max), min(my_max > 0 &&
my_min < max ? my_min : DEFAULT_MIN) {}
};
```

- Место делегированного конструктора первое в списке инициализации
- Далее делегирующий конструктор можно тоже делегировать и т.д.

ДЕСТРУКТОР

Аналогично созданию объекта при его уничтожении также вызывается особая функция-член, которая как раз и уничтожает объект – **деструктор**

Деструктор по объявлению схож с конструктором и предваряется знаком ~

```
class myPointer {
private:
    int* p;
public:
    ~myPointer() {delete [] p;} // dtor
};
```

ДЕСТРУКТОР

Деструктор не принимает никаких аргументов

Деструктор не может быть перегружен

АСИММЕТРИЯ ИНИЦИАЛИЗАЦИИ

Для класса с конструктором без аргументов нет разницы между

```
S s; // default-init, S()
S s{}; // default-init, S()
```

Но для примитивных типов и агрегатов разница гигантская

```
int n; // default-init, n = garbage
int m{}; // value-init, m = 0
int *p = new int[5]{} // calloc
```

То же самое для полей классов и т.д. рекурсивно

ПРОВЕРКА ЗРЕНИЯ

Что вы видите здесь?

```
struct Empty{



};
```

ПРОВЕРКА ЗРЕНИЯ

Что вы видите здесь?

```
struct Empty{



};
```

Программист видит возможность создать, скопировать и присвоить:

```
{ Empty x; Empty y(x); x = y; } // x, y destroyed
```

КОПИРОВАНИЕ И ПРИСВАИВАНИЕ

Копирование – это в основном способ инициализации

```
Copyable a;
Copyable b(a), c{a}; // прямое конструирование via copy ctor
Copyable d = a; // копирующее конструирование
```

Присваивание – это переписывание готового объекта

```
a = b; // присваивание
d = c =a = b; // присваивание цепочкой (правоассоциативно)
```

Ergo: Копирование **похоже на** конструктор. Присваивание совсем не похоже.

ПРОВЕРКА ЗРЕНИЯ С ЧУДО-ОЧКАМИ

Посмотрим на пустой класс через чудо-очки

```
struct Empty{
    Empty(); // ctor
    ~Empty(); // dtor
    Empty(const Empty&); // copy ctor
    Empty& operator=(const Empty&); // assignement
};
```

Все эти (и пару других) методы за вас написал компилятор

КЛАССЫ БЕЗ КОНСТРУКТОРА

Когда вы не определили ни одного конструктора, то компилятор определит за вас:

- конструктор по умолчанию;
- параметризованный конструктор (количество параметров и их типы определяются количеством и типом полей класса);
- конструктор копирования.

Если вы напишете хотя бы один из этих конструкторов, то компилятор остальные за вас создавать не будет (кроме конструктора копирования).

СЕМАНТИКА КОПИРОВАНИЯ

По умолчанию копирования и оператор присваивания реализуют:

- побитовое копирование и присваивание для встроенных типов и агрегатов
- Вызов конструктора копирования, если есть

```
struct Point2D {
    int x_, y_;
    Point2D() : default-init x_, default-init y_ {}
    ~Point2D() {}
    Point2D(const Point2D& rhs) : x_(rhs.x_), y_(rhs.y_) {}
    Point2D& operator=(const Point2D& rhs) {
        x_ = rhs.x_; y_ = rhs.y_; return *this;
    }
};
```

СЕМАНТИКА КОПИРОВАНИЯ

Должны ли мы неявное делать явным?

```
struct Point2D {
    int x_, y_;
};
```

- Здесь не нужны конструктор копирования и оператор присваивания
- По умолчанию копирование и присваивание отлично работает
- В таких случаях мы не должны определять копирование/присваивание

СЛУЧАЙ ОПАСНОСТИ НЕЯВНОГО

Кажется всё просто

```
class Buffer {
    int* p;
public:
    Buffer(int n) : p(new int[n]) {}
    ~Buffer() {delete [] p;}
};
```

Что может пойти не так?

СЛУЧАЙ ОПАСНОСТИ НЕЯВНОГО

Кажется всё просто

```
class Buffer {
    int* p;
public:
    Buffer(int n) : p(new int[n]) {}
    ~Buffer() {delete [] p;}
    Buffer(const Buffer& rhs) : p(rhs.p) {}
    Buffer& operator=(const Buffer& rhs) {p = rhs.p; …}
};
```

В Чудо-очках мы видим проблему:

```
{ Buffer x; Buffer y = x; } // double deletion
```

DEFAULT И DELETE

Мы можем явно попросить компилятор написать метод, написав `default` и явно запретить, написав `delete`

```
class Buffer {
    int* p;
public:
    Buffer(int n) : p(new int[n]) {}
    ~Buffer() {delete [] p;}
    Buffer(const Buffer& rhs) = delete
    Buffer& operator=(const Buffer& rhs) = delete
};

{ Buffer x; Buffer y = x; } // CE!
```

ВИДЫ КОНСТРУКТОРОВ

В зависимости от принимаемых аргументов конструкторы делятся на следующие:

- Конструктор не принимает никаких аргументов - Конструктор по умолчанию
- Конструктор принимает ровно один аргумент, отличный от самого класса – преобразующий конструктор
- Конструктор принимает более одного аргумента – параметризованный конструктор
- Конструктор принимает объект того же класса по константной lvalue ссылке – конструктор копирования
- Конструктор принимает объект того же класса по rvalue ссылке – конструктор перемещения

СИНТАКСИС ВЫЗОВА КОНСТРУКТОРОВ

```
struct S {
    int x = 0;
    int y = 0;
};
int main() {
    S s1; // default ctor
    S s2(10); // convert ctor
    S s3(10, 20); // param ctor
    S s4{1, 2}; // param ctor, initializer-list ctor
    S s5(); // NOT any ctor
    S s6 = S(); // default ctor
    S s7 = 10; // convert ctor
    S s8 = s7; // copy ctor
    S s9(s1); // copy ctor
}
```

РЕАЛИЗУЕМ КОПИРОВАНИЕ

```
class Buffer {
    int n_; int *p_;
public:
    Buffer(int n): n_(n), p_(new int[n_]) {}
    ~Buffer() {delete [] p_;}

    // думайте о «Buffer rhs; Buffer b{rhs};»
    Buffer(const Buffer& rhs) : n_(rhs.n_), p_(new int[n_]) {
        std::copy(p_, p_ + n_, rhs.p_)
    }

    Buffer& operator=(const Buffer& rhs);
};
```

РЕАЛИЗУЕМ ПРИСВАИВАНИЕ

```
Buffer& Buffer::operator=(const Buffer& rhs) {
    n_ = rhs.n_;
    delete [] p_;
    p_= new int[n_];
    std::copy(p_, p_ + n_, rhs.p_);
    return *this;
}
```

Тут можно визуализировать это как:

```
Buffer a, b; a = b;
```

Видите ли вы ошибку в коде?

НЕ ЗАБЫВАЕМ О СЕБЕ

```
Buffer& Buffer::operator=(const Buffer& rhs) {
    if (this = &rhs) return *this;
    n_ = rhs.n_;
    delete [] p_;
    p_= new int[n_];
    std::copy(p_, p_ + n_, rhs.p_);
    return *this;
}
```

Первая проблема – это присваивание вида a = a. Её довольно просто решить.

Вторая проблема сложнее. Её мы пока оставим и поговорим о специальной семантике копирования и присваивания.

RVO

```
struct foo {
    foo() {std::cout << "foo::foo()" << std::endl;}
    foo (const foo&) {std::cout << "foo::foo(const foo&)" <<
std::endl;}
    ~foo() {std::cout << "foo::~foo()" << std::endl;}
};

foo bar() {foo local_foo; return local_foo;}

int main() {
    foo f = bar();
    use(f);
    return 0;
}
```

Что должно быть на экране? А что реально будет?

ДОПУСТИМЫЕ ФОРМЫ

Поскольку конструктор копирования подвержен RVO, это не просто функция. У неё есть специальное значение, которое компилятор должен соблюдать.

Но чтобы он распознал конструктор копирования, у него должна быть одна из форм, предусмотренных стандартом. Основная форма – это константная ссылка.

```
struct Copyable {
    Copyable(const Copyable &c);
};
```

Допустимо также принимать неконстантную ссылку, как угодно **cv-квалифицированную** ссылку или значение.

CV-КВАЛИФИКАЦИЯ

В языке C++ есть два очень специальных квалификатора `const` и `volatile`.

Что означает `const` для объекта?

```
const int c = 34;
```

Что означает `volatile` для объекта?

```
volatile int v;
```

Что означает `const volatile` для объекта?

```
const volatile int cv = 42;
```

CV-КВАЛИФИКАЦИЯ

В языке C++ есть два очень специальных квалификатора `const` и `volatile`.

Что означает `const` для метода?

```
int S::foo() const { return 42; }
```

Что означает `volatile` для метода?

```
int S::bar() volatile { return 42; }
```

Что означает `const volatile` для метода?

```
int S::buz() const volatile { return 42; }
```

ИНТЕРЕСНЫЙ ОПЫТ

Что вы сможете сделать с `volatile` объектом `std::vector`?

```
volatile std::vector v;
```

Посмотрите предусмотренную стандартом реализацию.
Потом поэкспериментируйте самостоятельно.

ИНТЕРЕСНЫЙ ОПЫТ

Что вы сможете сделать с `volatile` объектом `std::vector`?

```
volatile std::vector v;
```

Посмотрите предусмотренную стандартом реализацию.
Потом поэкспериментируйте самостоятельно.

СПЕЦСЕМАНТИКА ИНИЦИАЛИЗАЦИИ

Обычные конструкторы определяют **неявное преобразование типа**.

```
struct MyString{
    char *buf_; size_t len_;
    MyString(size_t len) : buf_{new char[len]{}},
len_{len} {}
};
void foo(MyString);
foo(42); // ok, MyString implicitly constructed
```

Почти всегда это полезно

Но это **не всегда** хорошо, например в ситуации со строкой, мы ничего такого не имели ввиду.

ВНЕСЕМ ЯСНОСТЬ

Ключевое слово `explicit` указывается когда мы хотим заблокировать пользовательское преобразование

```
struct MyString{
    char *buf_; size_t len_;
    explicit MyString(size_t len) :
          buf_{new char[len]{}}, len_{len} {}
};
```

Теперь здесь ошибка компиляции

```
void foo(MyString);
foo(42); // error: could not convert ‘42’ from ‘int’
to ‘MyString’
```

DIRECT VS COPY

Важно понимать, что `explicit` конструкторы рассматриваются для прямой инициализации

```
struct Foo{
    explicit Foo(int x){} // блокирует неявные преобразования
};
Foo f{2}; // прямая инициализация
Foo f = 2; // инициализация копированием, FAIL
```

В этом смысле инициализация копированием похожа на вызов функции

ПОЛЬЗОВАТЕЛЬСКИЕ ПРЕОБРАЗОВАНИЯ

В некоторых случаях мы не можем сделать конструктор. Скажем, если мы хотим неявно преобразовать наш тип в `int`?
Тогда мы пишем `operator type`

```
struct MyString{
    char *buf_; size_t len_;
    /* explicit? */ operator const char*() {return buf;}
```

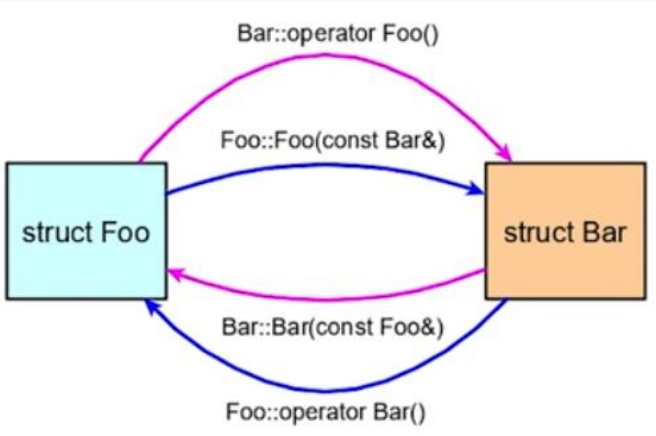
Можно `operator int`, `operator double`, `operator S` и так далее.
На такие операторы можно навешивать `explicit`, тогда возможно только явное преобразование

ПОЛЬЗОВАТЕЛЬСКИЕ ПРЕОБРАЗОВАНИЯ

Таким образом есть некая избыточность: два способа перегнать туда и два способа перегнать обратно

Конечно хороший тон – это использовать конструкторы, где возможно

Как вы думаете, что будет при конфликте?



ПЕРЕГРУЗКА

Пользовательские преобразования участвуют в перегрузке
Они проигрывают стандартным, но выигрывают у троеточий

```
struct Foo { Foo(long x = 0) {} };

void foo(int x);
void foo(Foo x);

void bar(Foo x);
void bar(…);
long lng; foo(lng); // вызовет foo(int)
bar(2); // вызовет bar(Foo);
```

ТАКИЕ РАЗНЫЕ ОПЕРАТОРЫ

Перегрузка операторов присваивания и приведения выглядит непохоже

```
struct Point2D {
    int x_, y_;
    Point2D& operator=(const Point2D& rhs) = default;
    operator int() {return x_;}
};
```

В мире конструкторов спецсемантика есть только у копирования и приведения

В мире переопределенных операторов она есть везде, и она нас ждёт уже совсем скоро!

ДОМАШНЯЯ РАБОТА

- Со стандартного ввода приходят ключи (каждый ключ – это целое число, **все ключи разные**) и запросы двух видов.
- Запрос (m) на поиск k-ого наименьшего элемента
- Запрос (n) на поиск количества элементов, меньших, чем заданный
- Вход: `k 8 k 2 k -1 m 1 m 2 n 3`
- Результат: `-1 2 2`
- Ключи могут быть как угодно перемешаны с запросами.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Дорохова Т.Ю., Основы алгоритмизации и программирования : учебное пособие для СПО / Т.Ю. Дорохова, И.Е. Ильина. – Саратов, Москва : Профобразование, Ай, Пи Ар Медиа, 2022. – 139 с.
2. Кудинов Ю.И., Основы алгоритмизации и программирования : учебное пособие для СПО / Ю.И. Кудинов, А.Ю. Келина. – 2-е изд. – Липецк, Саратов: Липецкий государственный технический университет, Профообразование, 2020. – 71 с.
3. Дональд Кнут, Искусство программирования. Том 1. Основные алгоритмы / Ю.В. Козаченко. - 3-е изд – Москва, Санкт-Петербург: ВИЛЬЯМС, 2018. – 721 с.