Зачем C++ математикам?

Соловьёв Владимир Валерьевич
Huawei, НГУ, СУНЦ
vladimir.conwor@gmail.com
vk.com/conwor

DSL!

Domain-specific languages

Языки, близкие к предметной области

Для конкретного класса задач, а не общего назначения

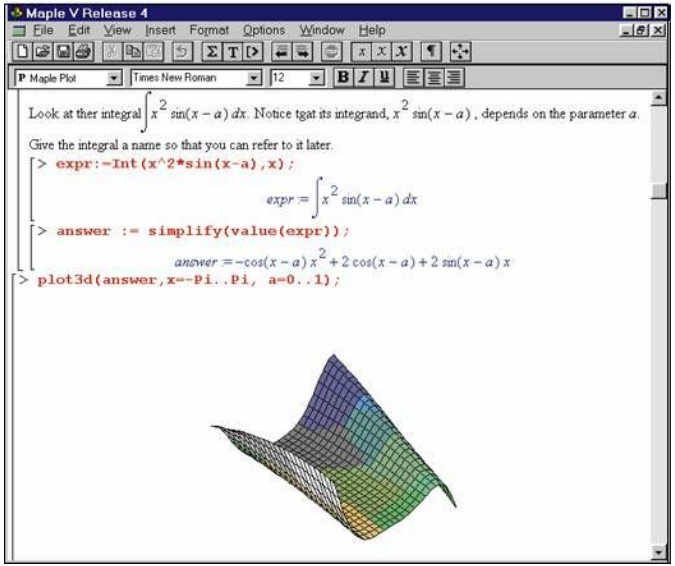
G-code

Язык программирования ЧПУ-станков


```
G00 X-25 Y20
G91 X-10
G02 X10 Y-10 I10 J0 F200 M3
G02 X-10 Y10 I0 J10;
G90 G00 M5 X0 Y0;
G00 X25 Y20
G91 X-10
G02 X10 Y-10 I10 J0 F200M3
G02 X-10 Y10 I0 J10;
G90 G00 M5 X0 Y0;
G00 X40 Y0
G02 X-40 Y0 I-40 J0 F200 M3
G90 G00 M5 X-55 Y0;
G02 X0 Y-55 I55 J0 F200 M3
G02 X-55 Y0 I0 J55
```

Maple/Mathematica

Математические вычисления, моделирование и визуализация



Prolog

Декларативный язык исчисления предикатов

```
parent("Tom","Jake").
parent("Janna","Jake").
parent("Tom","Tim").
male("Tom").
male("Tim").
male("Jake").
female("Janna").

brother(X,Y):-
parent(Z,X),parent(Z,Y),male(X),male(Y),X\=Y.
```

Языково-ориентированное программирование

Разделение задачи на подзадачи по областям знаний

Подбор или разработка DSL для областей

Решение подзадач на DSL

Embedded DSL

Реализованы внутри другого (базового) языка программирования

Возможности синтаксиса языка-носителя используются для создания DS-синтаксиса

Упрощается реализация и комбинирование с другими DSL в ЯОП-процессе

Инструменты EDSL в C++

Перегрузка операторов

Перегрузка операторов

```
class Vector {
        float* data;
        int size;
public:
        Vector add(Vector that);
};
```

```
Vector a;
Vector b;
…
Vector c = a.add(b);
```

Перегрузка операторов

```
class Vector {
        float* data;
        int size;
public:
        Vector add(Vector that);
};
```

```
Vector a;
Vector b;
…
Vector c = a + b;
```

Перегрузка операторов

```
class Vector {
    float* data;
    int size;
public:
    Vector(const Vector& that);
    Vector operator+ (const Vector& that) const;
};
```

```
Vector a;
Vector b;
…
Vector c = a + b;
```

Несколько мелких деталей

const, &, copy-constructor

Ключевое слово const

```
const int x = 42;

x = 37; // ошибка компиляции
```

```
const int x;
int const x;
```

одно и то же

Ключевое слово const на указателях

int*	-	указатель на int
const int*	-	указатель на неизменяемый int
int* const	-	неизменяемый указатель на int
const int* const	-	неизменяемый указатель на неизменяемый int

Ключевое слово const на указателях

int x, y;

	p = &y (изменение указателя)	*p = 37 (изменение значения по указателю)
int* p = &x		
const int* p = &x		
int* const p = &x		
const int* const p = &x		

корректная операция, ошибка компиляции

Ключевое слово const на указателях

int x, y;

	p = &y (изменение указателя)	*p = 37 (изменение значения по указателю)
int* p = &x		
const int* p = &x		
int* const p = &x		
const int* const p = &x		

корректная операция, ошибка компиляции

Ключевое слово const на указателях

Если переменная неизменяемая, то её адрес может быть записан только в указатель на неизменяемую сущность

Иначе неизменяемость переменной элементарно нарушить - присвоить адрес в обычный указатель и изменить

Ключевое слово const на указателях

```
const int x = 42;
int* p;

p = &x;
*p = 37; // ???
```

Ключевое слово const на указателях

```
const int x = 42;
int* p;

p = &x; // ошибка компиляции
*p = 37;
```

Ключевое слово const на указателях

p = &x;

	int x	const int x
int* p		
const int* p		

корректная операция, ошибка компиляции

Ключевое слово const на указателях

p = &x;

	int x	const int x
int* p		
const int* p		

корректная операция, ошибка компиляции

Ключевое слово const на указателях

Частный случай - передача адреса в функцию

Адрес неизменяемой переменной можно передать только в функцию, принимающую указатель на неизменяемую сущность

Ключевое слово const на указателях

foo(&x);

	int x	const int x
void foo(int* p)		
void foo(const int* p)		

корректная операция, ошибка компиляции

Ключевое слово const на указателях

foo(&x);

	int x	const int x
void foo(int* p)		
void foo(const int* p)		

корректная операция, ошибка компиляции

Ключевое слово const

foo(x);

	int x	const int x
void foo(int p)		
void foo(const int p)		

корректная операция, ошибка компиляции

Ключевое слово const

foo(x);

	int x	const int x
void foo(int p)		
void foo(const int p)		

корректная операция, ошибка компиляции

Ключевое слово const

```
void foo(int p) {
        ...
}


const int x;
foo(x);
```

p - **<u>копия</u>** x
const оригинала и const копии никак не связаны

Ключевое слово const

```
class Foo {
public:
	int field;
};
```

```
const Foo f;
f.field = 37; // ошибка компиляции!
```

Ключевое слово const

```
class Foo {
public:
	int field;
};
```

```
const Foo f; // ошибка компиляции!
f.field = 37; // ошибка компиляции!
```

Ключевое слово const

```
class Foo {
public:
	int field;
};
```

```
const Foo f; // ошибка компиляции!
f.field = 37; // ошибка компиляции
```

поле field нельзя будет потом изменить
значит, оно должно быть проинициализировано сразу
но конструктор по умолчанию этого не делает!

Ключевое слово const

```
class Foo {
public:
        int field;

        Foo() {
                this->field = 42;
        }
};
```

```
const Foo f;
```

Ключевое слово const

```
class Foo {
public:
        int field;
        Foo() {... }

        void bar() {
                …
        }
};
```

```
const Foo f;
f.bar();
```

Ключевое слово const

```
class Foo {
public:
        int field;
        Foo() {...}

        void bar() {
                …
        }
};
```

```
const Foo f;
f.bar(); // ошибка компиляции
```

Ключевое слово const

```
class Foo {
public:
        int field;
        Foo() {...}

        void bar(/* Foo* this */) {
                …
        }
};
```

```
const Foo f;
f.bar(); // ошибка компиляции
```

f - обычный параметр своего метода
на него действуют const-ограничения

Ключевое слово const

```
class Foo {
public:
        int field;
        Foo() {...}

        void bar() const {
                …
        }
};
```

```
const Foo f;
f.bar();
```

const-метод
Аргумент this передаётся, как const Foo*

Ключевое слово const

```
class Foo {
public:
        int field;
        Foo() {...}

        void bar() const {
                this->field = 37; // ошибка компиляции
        }
};
```

```
const Foo f;
f.bar();
```

const-метод
Аргумент this передаётся, как const Foo*

const на полях

```
class Foo {
public:
        const int field;
};
```

```
Foo f;
```

const на полях

```
class Foo {
public:
        const int field;
};
```

```
Foo f; // ошибка компиляции
```

const на полях

```
class Foo {
public:
        const int field;
};
```

```
Foo f; // ошибка компиляции
```

Поле field не инициализировано

const на полях

```
class Foo {
public:
        const int field;
};
```

```
Foo f; // ошибка компиляции
```

Поле field не инициализировано, **и его уже негде инициализировать!**

const на полях

```
class Foo {
public:
	const int field;

	Foo() {
		this->field = 37;
	}
};
```

```
Foo f; // ошибка компиляции
```

Поле field не инициализировано, **и его уже негде инициализировать!**

const на полях

```
class Foo {
public:
	const int field;

	Foo() {
		this->field = 37;
	}
};
```

```
Foo f; // ошибка компиляции
```

Поле field не инициализировано, **и его уже негде инициализировать!**

const на полях

```
class Foo {
public:
        const int field;

        Foo(): field(37) {
        }
};
```

```
Foo f;
```

const на полях

```
class Foo {
public:
        const int field = 37;

        Foo() {
        }
};
```

```
Foo f;
```

const на полях

```
class Foo {
public:
        const int field = 37;

        Foo() {
        }

        Foo(int f): field(f) {
        }
};
```

```
Foo f;
```

33 слайда об одном ключевом слове

Но суть проста:

1) Константы должны быть инициализированы один раз в жизни, и до любого использования
2) На константу может указывать только указатель на константу

Ссылки

Это указатели

Ссылки

Это указатели (почти)

1) Не изменяются (неявный const)
2) Чтение/запись - с неявным разыменованием
3) Не конвертируются в числа и обратно
4) Нет арифметики ссылок, нет NULL (без хаков)

У ссылок нет значений в обычном понимании этого слова, т.к. синтаксис не позволяет эти значения никаким образом получить

Ссылки

```
int x, y;
int* p = &x;

*p = 42;
y = *p;

p = &y;
```

Ссылки

```
int x, y;
int* p = &x;

*p = 42;
y = *p;

p = &y;
```

```
int x, y;
int& r = x; // заводим ссылку (и тут же инициализируем)
```

Ссылки

```
int x, y;
int* p = &x;

*p = 42;
y = *p;

p = &y;
```

```
int x, y;
int& r = x; // заводим ссылку (и тут же инициализируем)

r = 42; // запись по ссылке
y = r; // и чтение из неё включает неявное разыменование
```

Ссылки

```
int x, y;
int* p = &x;

*p = 42;
y = *p;

p = &y;
```

```
int x, y;
int& r = x; // заводим ссылку (и тут же инициализируем)

r = 42; // запись по ссылке
y = r; // и чтение из неё включает неявное разыменование

r = y; // присваивание y в x, переставить ссылку нельзя
```

Передача аргумента по ссылке

```
void foo(int& r) {
	std::cout << r;
}

int x = 42;
foo(x); // выведется 42
```

Передача аргумента по ссылке

```
void foo(int& r) {
	std::cout << r;
}

int x = 42;
foo(x); // выведется 42

foo(5); // ошибка компиляции, также, как и попытка взять адрес числа 5
```

Передача аргумента по ссылке

```
void foo(int& r) {
	std::cout << r;
}

int x = 42;
foo(x); // выведется 42

foo(5); // ошибка компиляции, также, как и попытка взять адрес числа 5

int* p = NULL;
foo(*p); // segmentation fault
```

Передача аргумента по ссылке

```
void foo(int& r) {
        r = 37;
}

int x = 42;
foo(x);
// после вызова x будет равен 37
```

Передача по ссылке подобна передаче по указателю языка C

Передача аргумента по ссылке

```
struct Foo {
        int x;
        int y;
        int z;

        Foo() {...}
};
```

```
void bar(Foo& f) {
        f.x = 42;
}
```

```
Foo obj;
obj.x = 37;
bar(obj);
// после вызова obj.x будет равен 42
```

Передача аргумента по ссылке

```
struct Foo {
        int x;
        int y;
        int z;

        Foo() {...}
};
```

```
void bar(Foo& f) {
        ...
}


const Foo obj;
bar(obj); // ошибка компиляции
```

Передача аргумента по ссылке

```
struct Foo {
        int x;
        int y;
        int z;

        Foo() {...}
};
```

```
void bar(const Foo& f) {
        ...
}


const Foo obj;
bar(obj);
```

Передача по ссылке - неявная семантика

Язык C

```
int x = 1, y = 2, z = 3;
foo(x, y, &z);
// x - 1, y - 2, z - ???
```

Язык C++

```
int x = 1, y = 2, z = 3;
foo(x, y, &z);
// x - ???, y - ???, z - ???
```

Передача по ссылке - неявная семантика

Язык C

```
int x = 1, y = 2, z = 3;
foo(x, y, &z);
// x - 1, y - 2, z - ???
```

Язык C++

```
int x = 1, y = 2, z = 3;
foo(x, y, &z);
// x - ???, y - ???, z - ???
```

```
void foo(int& p1, int p2, int* p3) {
    p1 = 42;
}
```

Передача по ссылке

1) Для экономии действий (копируется адрес структуры, а не сама структура)
2) Для изменения переданных параметров
3) Изменяемость параметра контролируется спецификатором const
4) В отличие от C, нельзя по виду вызова судить о неизменяемости переданных переменных, нужно смотреть описание метода

Конструктор копирования

```
class Vector {
        float* data;
        int size;
public:
        Vector(int size) {
                this->data = new int[size];
                this->size = size;
        }

        ~Vector() {
                delete[] this->data;
        }
}
```

```
void foo(Vector v) {
        ...
}


Vector x(8);
foo(x);
```

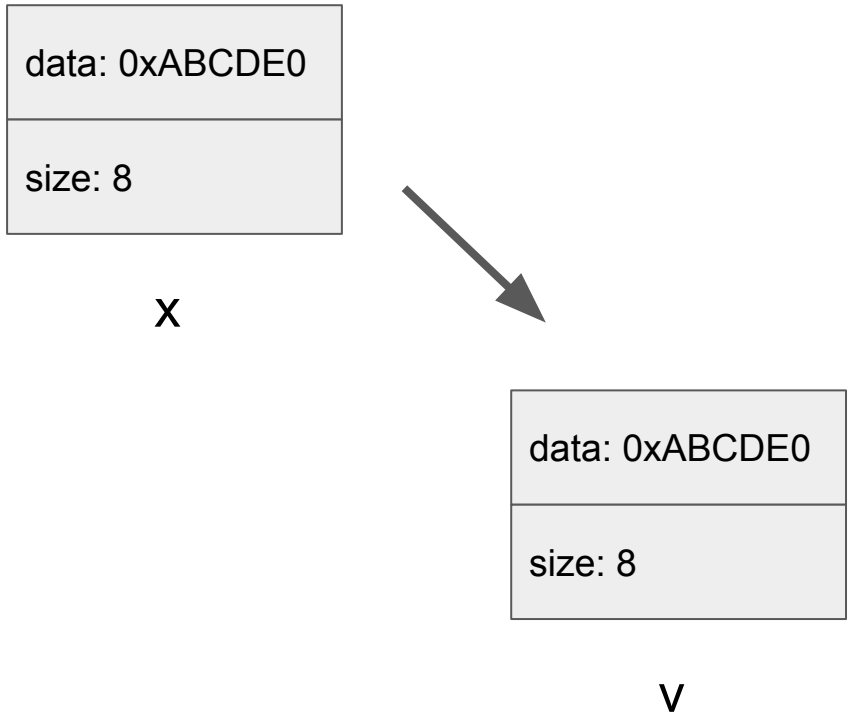
Конструктор копирования

```
void foo(Vector v) {
        ...
}


Vector x(8);
foo(x);
```

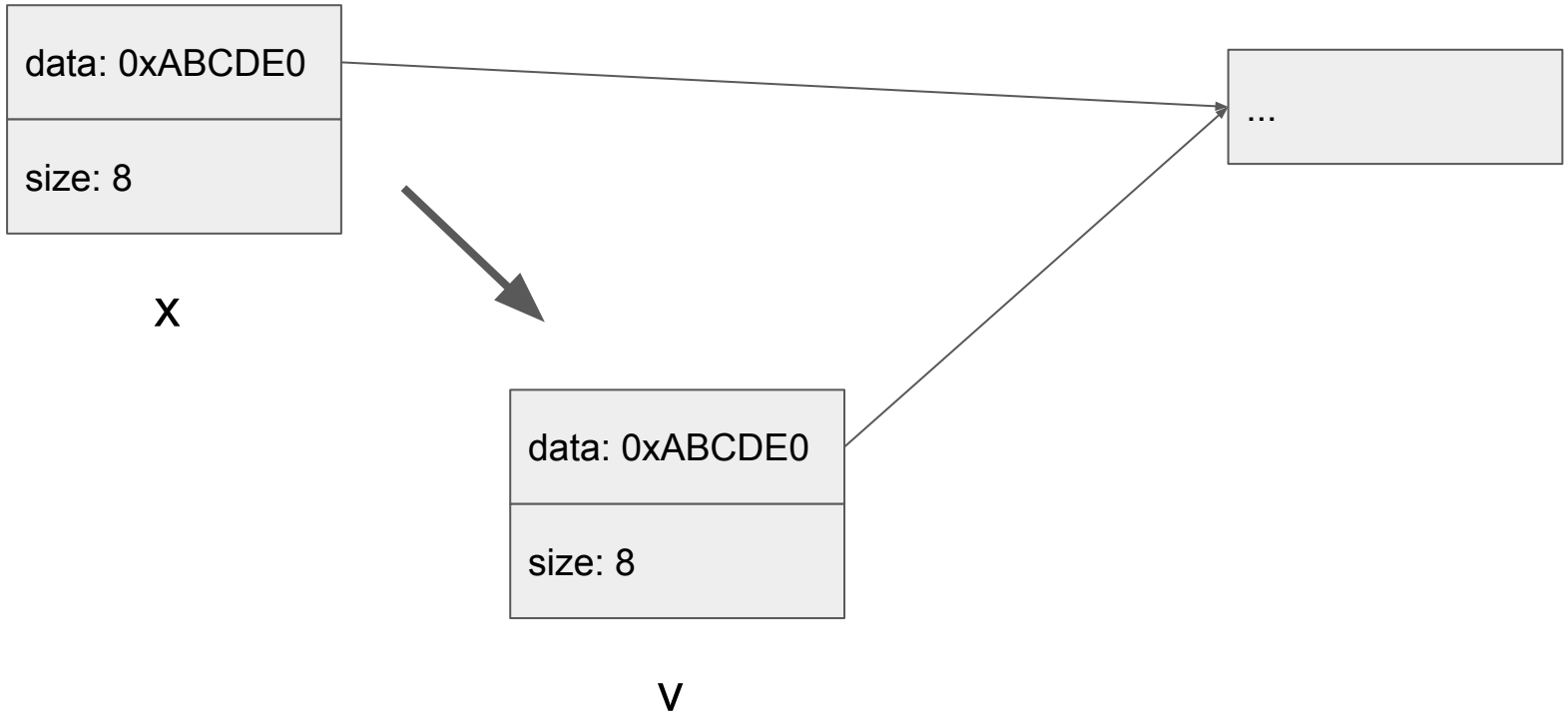
v - копия x

копирование по умолчанию - побайтовое, без углубления по адресам

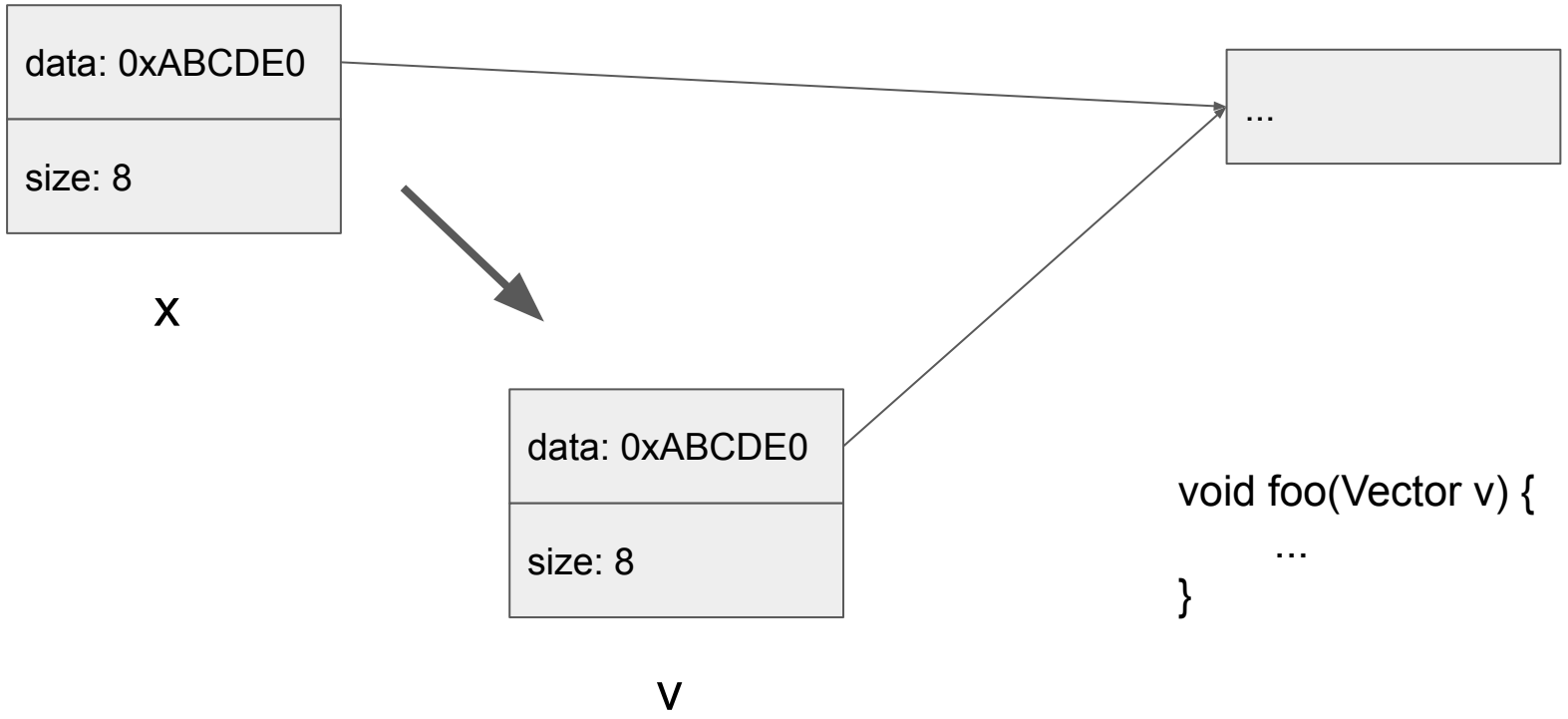
Конструктор копирования



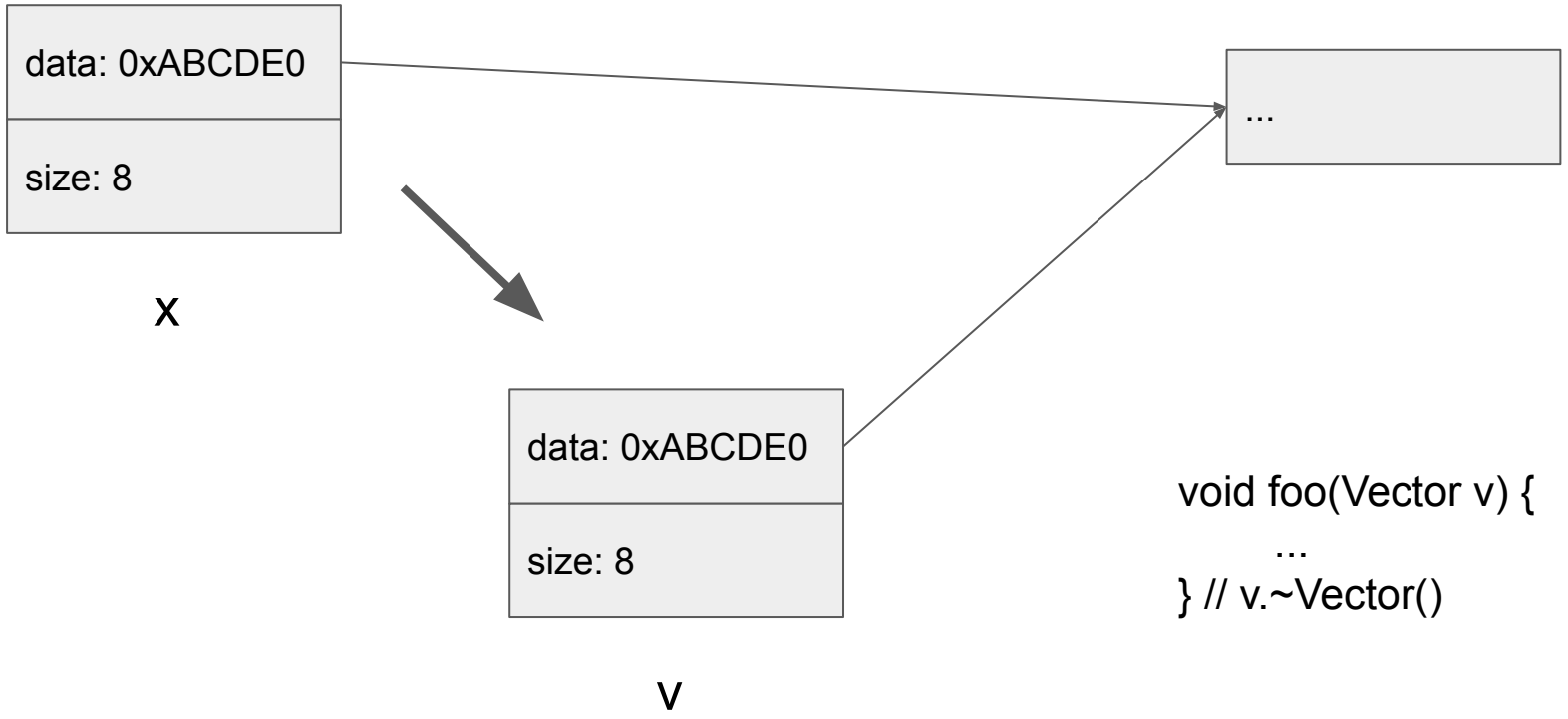
Конструктор копирования



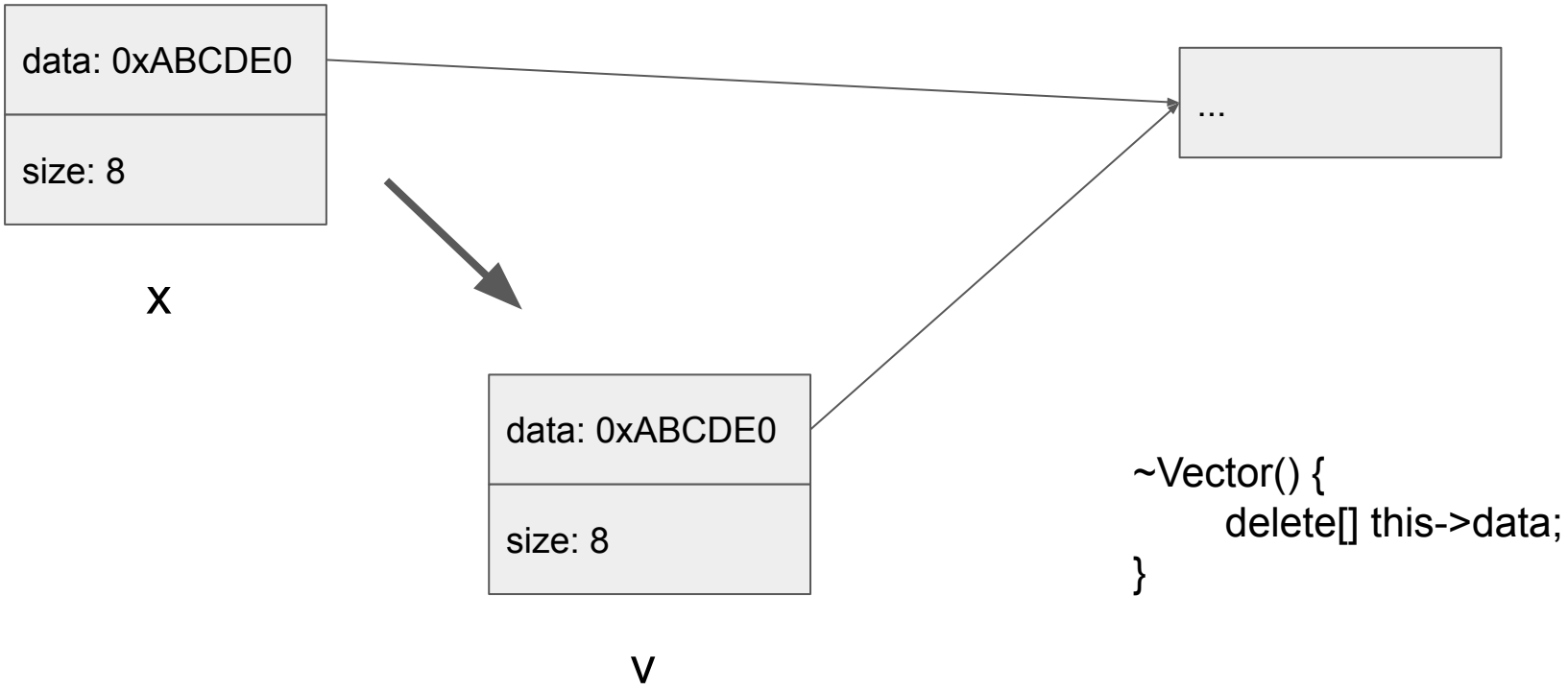
Конструктор копирования



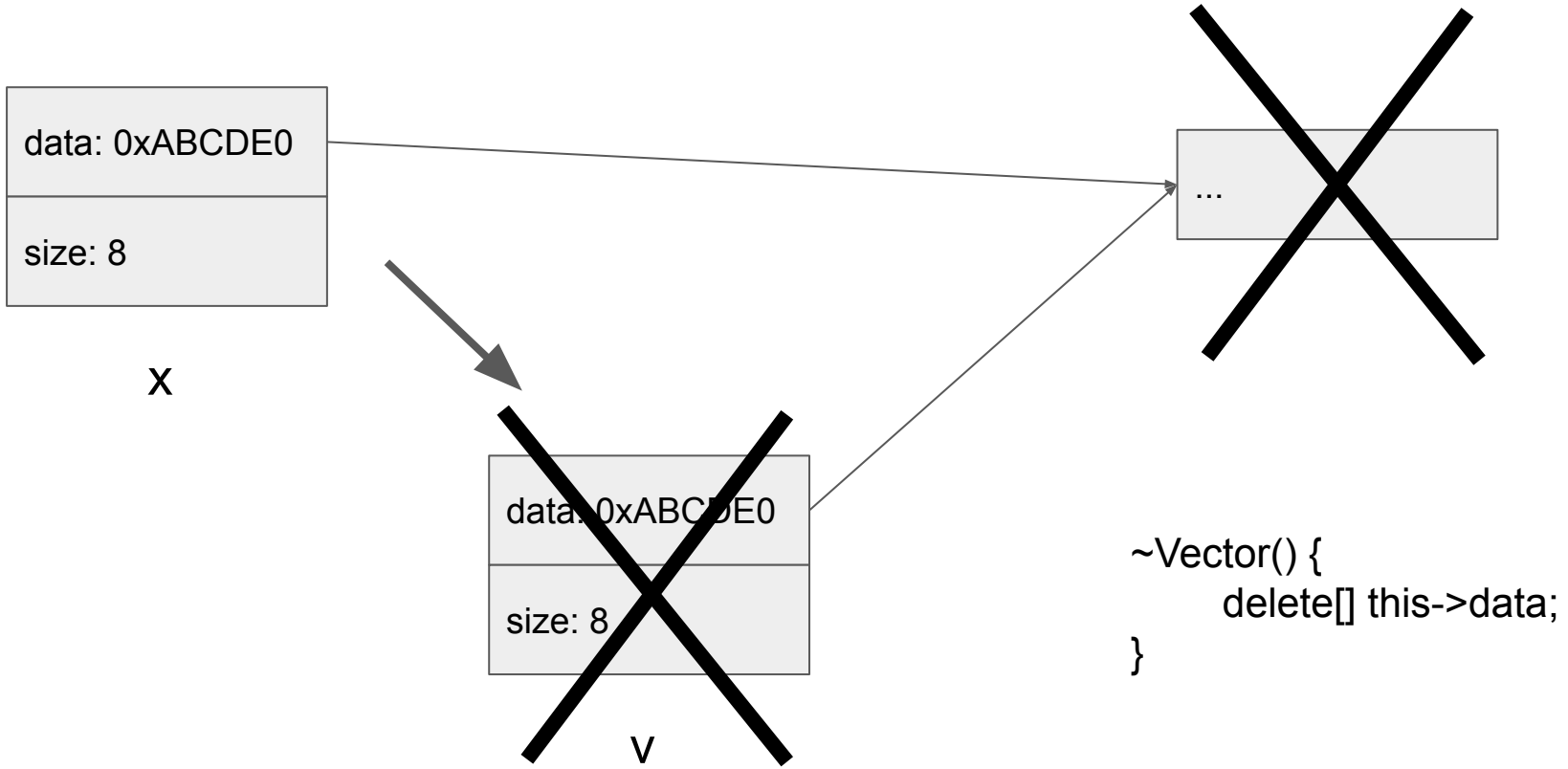
Конструктор копирования



Конструктор копирования



Конструктор копирования



Конструктор копирования по умолчанию

Побайтово, без углубления по адресам

Подходит для простых “однослойных” структур, не владеющих памятью, на которую указывают

Конструктор копирования по умолчанию

```
class Point {
        float x;
        float y;
        float z;
public:
        ...
};
```

```
class LineSegment {
        Point* a;
        Point* b;
public:
        LineSegment(Point* a, Point* b) {
                this->a = a;
                this->b = b;
        }

        ~LineSegment() {
        }
};
```

Конструктор копирования

Для более сложных типов нужно определять свой способ копирования

```
class Vector {
        …
        Vector(const Vector& that) {
                this->size = that.size;
                this->data = new int[this->size];
                for (int i = 0; i < this->size; i++) {
                        this->data[i] = that.data[i];
                }
        }
};
```

Конструктор копирования

Вызывается в трёх случаях:

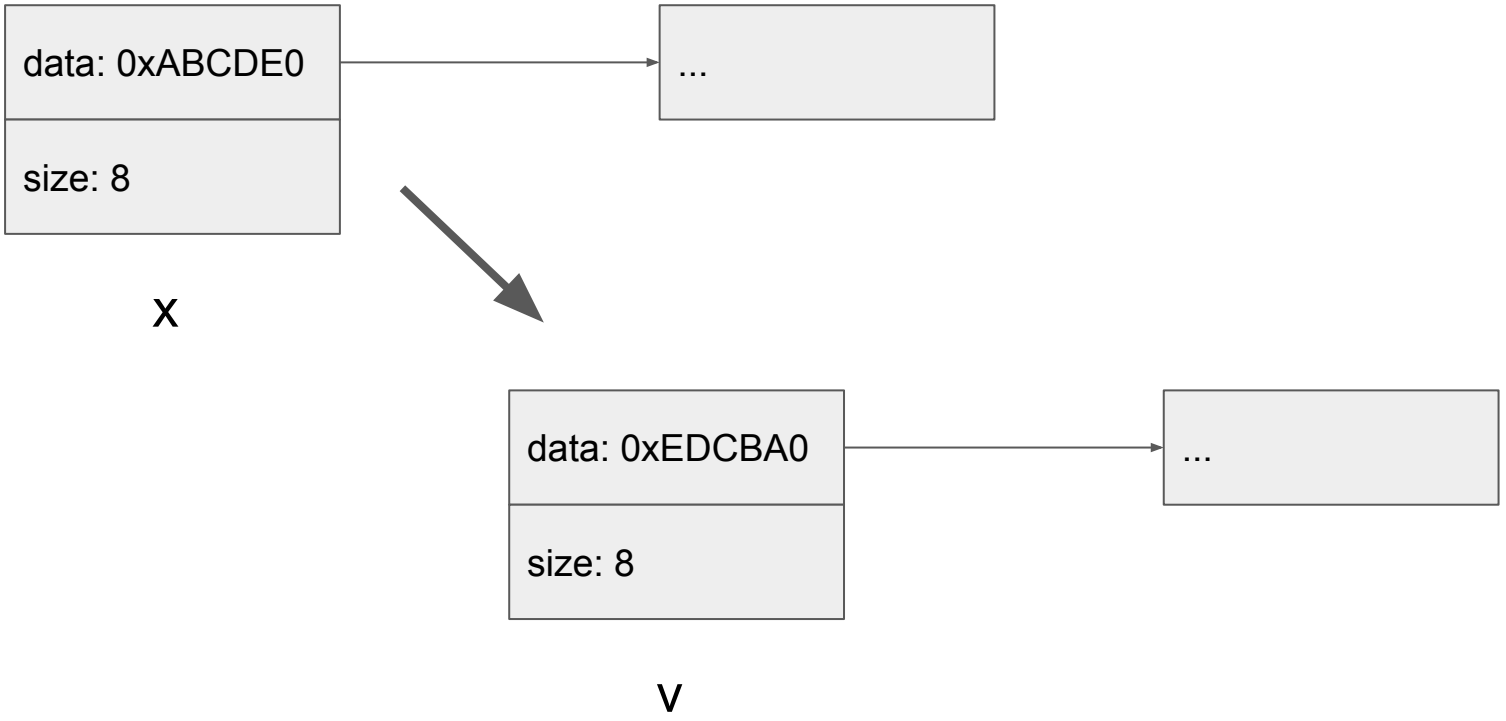
1) Создание нового объекта (явно)
2) Передача в функцию по значению
3) Возврат значения из функции

```
Vector x(8);
Vector y(x);
Vector z = x;
```

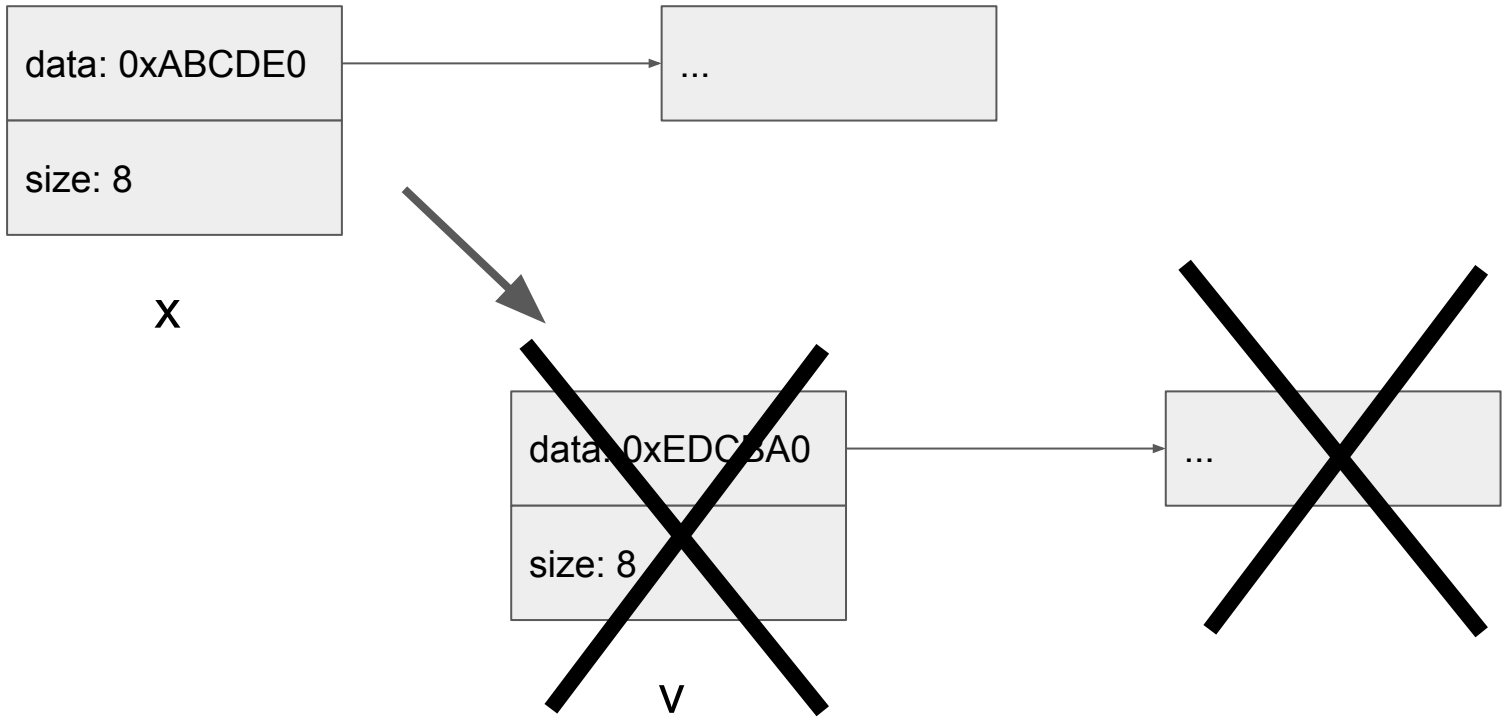
```
void foo(Vector v) {...}

Vector x(8);
foo(x);
```

```
Vector bar() {
    Vector v(8);
    …
    return v;
}
```

Конструктор копирования



Конструктор копирования



Конструктор копирования

```
class Vector {
        …
        Vector(const Vector& that) {
                this->size = that.size;
                this->data = new int[this->size];
                for (int i = 0; i < this->size; i++) {
                        this->data[i] = that.data[i];
                }
        }
};
```

Конструктор копирования должен принимать свой аргумент по ссылке

Иначе его бы пришлось как-то скопировать (и напороться на те же проблемы)

Конструктор копирования

```
class Vector {
        …
        Vector(const Vector& that) {...}
};
```

Конструктор копирования

```
class Vector {
        …
        Vector(/* const */ Vector& that) {...}
};
```

```
const Vector v(8);
Vector v2 = v; // ошибка компиляции
```

Конструктор копирования

```
class Vector {
        …
        Vector(/* const */ Vector& that) {...}
};
```

Временные объекты (результаты вызова функций, вычислений, ...) константны

```
Vector foo() { … }

Vector x = foo(); // ошибка компиляции
```

Конструктор копирования

```
class Vector {
        …
        Vector(const Vector& that) {...}
};
```

const можно не указывать, но это неприятно ограничивает область применения, так что смысла нет.

Кроме классов, копирование которых невозможно без модификации оригинала.

упражнение - придумать адекватный пример

О чём мы там говорили?

Перегрузка операторов

```
class Vector {
        float* data;
        int size;
public:
        Vector(const Vector& that);
        Vector operator+ (const Vector& that) const;
};
```

Перегрузка операторов

```
class Vector {
    ...
    Vector operator+ (const Vector& that) const {
        Vector result(this->size);
        for (int i = 0; i < this->size; i++) {
            result.data[i] = this->data[i] + that.data[i];
        }
        return result;
    }
};
```

Перегрузка операторов

```
class Vector {
    ...
    Vector operator+ (const Vector& that) const {
        Vector result(this->size);
        for (int i = 0; i < this->size; i++) {
            result.data[i] = this->data[i] + that.data[i];
        }
        return result;
    }
};
```

для эффективности
(не вызывать КК)

Перегрузка операторов

```
class Vector {
    ...
    Vector operator+ (const Vector& that) const {
        Vector result(this->size);
        for (int i = 0; i < this->size; i++) {
            result.data[i] = this->data[i] + that.data[i];
        }
        return result;
    }
};
```

для расширения области применения
(второй аргумент может быть константным)

Перегрузка операторов

```
class Vector {
    ...
    Vector operator+ (const Vector& that) const {
        Vector result(this->size);
        for (int i = 0; i < this->size; i++) {
            result.data[i] = this->data[i] + that.data[i];
        }
        return result;
    }
};
```

для расширения области применения
(первый аргумент может быть константным)

Перегрузка операторов

```
class Vector {
    ...
    Vector operator+ (const Vector& that) const {
        Vector result(this->size);
        for (int i = 0; i < this->size; i++) {
            result.data[i] = this->data[i] + that.data[i];
        }
        return result;
    }
};
```

Перегрузка операторов

```
class Vector {
    ...
    Vector& operator+ (const Vector& that) const {
        Vector result(this->size);
        for (int i = 0; i < this->size; i++) {
            result.data[i] = this->data[i] + that.data[i];
        }
        return result;
    }
};
```

Перегрузка операторов

```
class Vector {
    ...
    Vector& operator+ (const Vector& that) const {
        Vector result(this->size);
        for (int i = 0; i < this->size; i++) {
            result.data[i] = this->data[i] + that.data[i];
        }
        return result;
    } // result.~Vector()
};
```

ссылку на локальный объект возвращать нельзя

также, как и адрес локальной переменной

Что можно ещё перегрузить?

Все операторы, определённые в C++, кроме

:: . .* ?: sizeof typeid

При перегрузке && и || теряется свойство ленивости

Перегрузка операторов

```
class Vector {
        ...
        float operator[] (int index) const {
                return this->data[index];
        }

        Vector operator+ (const Vector& that) const {
                Vector result(this->size);
                for (int i = 0; i < this->size; i++) {
                        result.data[i] = (*this)[i] + that[i];
                }
                return result;
        }
};
```

ссылки удобнее использовать с операторами, чем указатели

Перегрузка операторов

```
class Vector {
        ...
        float operator[] (int index) const {
                return this->data[index];
        }

        Vector operator+ (const Vector& that) const {
                Vector result(this->size);
                for (int i = 0; i < this->size; i++) {
                        result.data[i] = (*this)[i] + that[i];
                }
                return result;
        }
};
```

а можно ли здесь применить оператор [] ?

Перегрузка операторов

```
class Vector {
        ...
        float operator[] (int index) const {
                return this->data[index];
        }

        Vector operator+ (const Vector& that) const {
                Vector result(this->size);
                for (int i = 0; i < this->size; i++) {
                        result[i] = (*this)[i] + that[i];
                }
                return result;
        }
};
```

а можно ли здесь применить оператор [] ?

Перегрузка операторов

```
class Vector {
        ...
        float operator[] (int index) const {
                return this->data[index];
        }

        Vector operator+ (const Vector& that) const {
                Vector result(this->size);
                for (int i = 0; i < this->size; i++) {
                        result[i] = (*this)[i] + that[i];
                }
                return result;
        }
};
```

а можно ли здесь применить оператор [] ?

lvalue и rvalue

lvalue - что-то, у чего есть определённый адрес

переменная (в том числе неизменяемая), ссылка, указатель с разыменованием

rvalue - всё остальное

литерал, результат выражения и т.п., не являющееся lvalue

lvalue и rvalue

В левой части оператора присваивания может стоять только lvalue

Перегрузка операторов

```
class Vector {
    ...
    float operator[] (int index) const { // возвращается rvalue
        return this->data[index];
    }

    Vector operator+ (const Vector& that) const {
        Vector result(this->size);
        for (int i = 0; i < this->size; i++) {
            result[i] = (*this)[i] + that[i];
        }
        return result;
    }
};
```

Перегрузка операторов

```
class Vector {
        ...
        float operator[] (int index) const { return this->data[index]; }

        float& operator[] (int index) { return this->data[index]; }

        Vector operator+ (const Vector& that) const {
                ...
                result[i] = (*this)[i] + that[i];
                ...
        }
};
```

Перегрузка операторов

```
class Vector {
        ...
        float operator[] (int index) const { return this->data[index]; }

        float& operator[] (int index) { return this->data[index]; }

        Vector operator+ (const Vector& that) const {
                ...
                result[i] = (*this)[i] + that[i];
                ...
        }
};
```

а вот это уже lvalue!

Перегрузка операторов

```
class Vector {
        ...
        float operator[] (int index) const { return this->data[index]; }

        float& operator[] (int index) { return this->data[index]; }

        Vector operator+ (const Vector& that) const {
                ...
                result[i] = (*this)[i] + that[i];
                ...
        }
};
```

но его нельзя вызывать от константных объектов

Перегрузка операторов

```
class Vector {
        ...
        float operator[] (int index) const { return this->data[index]; }

        float& operator[] (int index) { return this->data[index]; }

        Vector operator+ (const Vector& that) const {
                ...
                result[i] = (*this)[i] + that[i];
                ...
        }
};
```

компилятор догадается, кого позвать в таком случае

Оператор присваивания

В отличии от конструктора копий работает с уже созданным объектом

- Должен сначала счистить старое состояние
- Потом скопировать оригинал

Оператор присваивания

В отличии от конструктора копий работает с уже созданным объектом

- Должен сначала счистить старое состояние (деструктор)
- Потом скопировать оригинал (конструктор копирования)

Оператор присваивания

```
class Vector {
private:
	void rawClean() { delete[] this->data; }
	void rawCopy(const Vector& that) { this->size = that->size; this->data = new ... }
public:
	Vector(const Vector& that) { this->rawCopy(that); }
	~Vector() { this->rawClean(); }

	Vector& operator= (const Vector& that) {
		if (this != &that) {
			this->rawClean(); this->rawCopy(that);
		}
		return *this;
	}
};
```

Оператор присваивания по умолчанию

Побайтовый, без углубления по адресам

Не подходит для сложных структур с дополнительной памятью

Правило трёх

Если в классе есть один из:

- Деструктор
- Конструктор копирования
- Оператор присваивания

То в нём должны быть оставшиеся два

Оператор присваивания

```
class Vector {
        ...
        Vector& operator= (const Vector& that) {
                if (this != &that) {
                        this->rawClean(); this->rawCopy(that);
                }
                return *this;
        }
};
```

для эффективности
(не вызывать КК)

Оператор присваивания

```
class Vector {
        ...
        Vector& operator= (const Vector& that) {
                if (this != &that) {
                        this->rawClean(); this->rawCopy(that);
                }
                return *this;
        }
};
```

для расширения области применения
(второй аргумент может быть константным)

Оператор присваивания

```
class Vector {
        ...
        Vector& operator= (const Vector& that) {
                if (this != &that) {
                        this->rawClean(); this->rawCopy(that);
                }
                return *this;
        }
};
```

для защиты от самокопирования

```
Vector a(8);
Vector& r = a;
…
a = r;
```

Оператор присваивания

```
class Vector {
        ...
        Vector& operator= (const Vector& that) {
                if (this != &that) {
                        this->rawClean(); this->rawCopy(that);
                }
                return *this;
        }
};
```

lvalue - для применения
в цепочках присваиваний

```
Vector a(8);
Vector b(4);
Vector c(16);
…
a = b = c;
```

Оператор присваивания

```
class Vector {
        ...
        Vector& operator= (const Vector& that) {
                if (this != &that) {
                        this->rawClean(); this->rawCopy(that);
                }
                return *this;
        }
};
```

lvalue - для применения
в цепочках присваиваний

```
Vector a(8);
Vector b(4);
Vector c(16);
…
(a = b) = c;
```

Не путать

```
Vector a(8);

Vector b = a;        // конструктор копий

Vector c(6);
c = a;               // оператор присваивания
```

Перегрузка операторов

```
class Vector {
        …
        Vector& operator+= (const Vector& that) {
                *this = *this + that;
                return *this;
        }
};
```

упражнение - определить всю возникающую при исполнении неявную семантику:
временные объекты, конструкторы копий, деструкторы

Подводные камни

Эффективность (из-за неявной семантики и возврата по значениям)

Запутанность (из-за придания операторам неочевидных свойств)

- Например, побочные эффекты в операции сложения векторов
- Или оператор !, вычисляющий жорданову форму матрицы
- Или оператор -, складывающий вектора

Убедитесь, что вынесли с этой лекции

DSL/EDSL

Ключевое слово const

Ссылки

Конструктор копирования

Перегрузка операторов

lvalue / rvalue

Правило трёх

Проверочные вопросы

1) Чем отличаются ссылки от указателей?
2) В каких случаях вызывается конструктор копирования?
3) Какие минусы у оператора присваивания по умолчанию?
4) Почему конструктор копирования принимает аргумент по ссылке?
5) Можно ли в конструктор копирования передать константный объект?

*

1) Выполнить упражнения на слайдах №82 и №115
2) Оптимизировать пример на слайде №115, измерить разницу времени исполнения

Q & A