

Программирование—2

01 | Зачем C++ математикам?

20 сентября 2022

Зачем C++ математикам?

- DSL!

Domain-specific languages

- Языки, близкие к предметной области
- Для конкретного класса задач, а не общего назначения

G-code

Язык программирования ЧПУ-станков

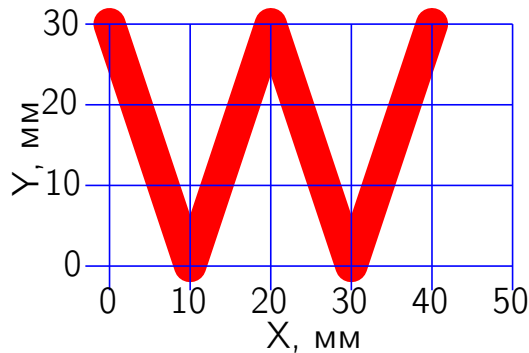
Язык программирования ЧПУ-станков

```
%  
O200  
G21 G40 G49 G53 G80 G90 G17  
G0 F300  
M3 S500  
G4 P2000  
X0 Y30 Z5  
G1 Z-2 F40  
G1 F20 X10 Y0  
X20 Y30  
X30 Y0  
X40 Y30  
G0 Z5  
M5  
M30
```

G-code

Язык программирования ЧПУ-станков

```
%  
O200  
G21 G40 G49 G53 G80 G90 G17  
G0 F300  
M3 S500  
G4 P2000  
X0 Y30 Z5  
G1 Z-2 F40  
G1 F20 X10 Y0  
X20 Y30  
X30 Y0  
X40 Y30  
G0 Z5  
M5  
M30
```



Maple/Mathematica

Математические вычисления, моделирование и визуализация

Maple/Mathematica

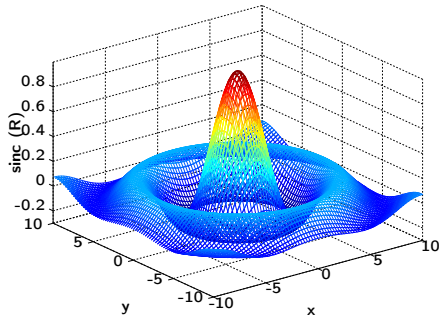
Математические вычисления, моделирование и визуализация

```
[X,Y] = meshgrid(-10:0.25:10,-10:0.25:10);  
f = sinc(sqrt((X/pi).^2+(Y/pi).^2));  
mesh(X,Y,f);  
axis([-10 10 -10 10 -0.3 1])  
xlabel('{\bf x}')ylabel('{\bf y}')zlabel('{\bf sinc} ({\bf R})')hidden off
```


Maple/Mathematica

Математические вычисления, моделирование и визуализация

```
[X,Y] = meshgrid(-10:0.25:10,-10:0.25:10);  
f = sinc(sqrt((X/pi).^2+(Y/pi).^2));  
mesh(X,Y,f);  
axis([-10 10 -10 10 -0.3 1])  
xlabel('\bfx')  
ylabel('\bfy')  
zlabel('\bfsinc ({\bfR})')  
hidden off
```



Prolog

Декларативный язык исчисления предикатов

Prolog

Декларативный язык исчисления предикатов

```
partition([], _, [], []).  
partition([X|Xs], Pivot, Smalls, Bigs) :-  
    (   X @< Pivot ->  
        Smalls = [XRest],  
        partition(Xs, Pivot, Rest, Bigs)  
    ;   Bigs = [XRest],  
        partition(Xs, Pivot, Smalls, Rest)  
    ).
```

```
quicksort([])    --> [].  
quicksort([X|Xs]) -->  
    { partition(Xs, X, Smaller, Bigger) },  
    quicksort(Smaller), [X], quicksort(Bigger).
```

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

Инструменты EDSL в C++

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

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

Инструменты EDSL в C++

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

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


Инструменты EDSL в C++

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

```
Vector a;  
Vector b;  
Vector c = a + (b);
```

Инструменты EDSL в C++

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

Несколько мелких деталей

- `const`
- `&`
- `copy-constructor`

Ключевое слово const

- Защищает значение от изменений

Ключевое слово const

- Защищает значение от изменений

```
const int x = 42;
```

```
x = 37; // ошибка компиляции
```

Ключевое слово const

- Защищает значение от изменений

```
const int x = 42;  
x = 37; // ошибка компиляции
```

- Одно и то же:

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

Ключевое слово `const` на указателях

<code>int *</code>	— указатель на <code>int</code>
<code>int const *</code>	— указатель на неизменяемый <code>int</code>
<code>const int *</code>	
<code>int * const</code>	— неизменяемый указатель на <code>int</code>
<code>const int * const</code>	— неизменяемый указатель на неизменяемый <code>int</code>

Ключевое слово const на указателях

Указатель на неизменяемый int

`int const` * `p`;

указатель

тип значения
на которое указывает

Ключевое слово const на указателях

Неизменяемый указатель на int


`int * const p;`
тип значения
на которое указывает

Ключевое слово const на указателях

```
int x, y;
```

	<code>p = &y;</code> (изменение указателя)	<code>*p = 37;</code> (изменение значения)
<code>int * p = &x</code>		
<code>int const * p = &x</code>		
<code>int * const p = &x</code>		
<code>int const * const p = &x</code>		

Ключевое слово const на указателях

```
int x, y;
```

	p = &y; (изменение указателя)	*p = 37; (изменение значения)
int * p = &x	✓	✓
int const * p = &x	✓	✗
int * const p = &x	✗	✓
int const * const p = &x	✗	✗

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

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

Ключевое слово const на указателях

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

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

Ключевое слово const на указателях

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

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

Ключевое слово `const` на указателях

	<code>int x</code>	<code>int const x</code>
<code>int * p = &x;</code>		
<code>int const * p = &x;</code>		

Ключевое слово `const` на указателях

	<code>int x</code>	<code>int const x</code>
<code>int * p = &x;</code>	✓	✗
<code>int const * p = &x;</code>	✓	✓

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)`

	<code>int x</code>	<code>int const x</code>
<code>void foo(int * p)</code>		
<code>void foo(int const * p)</code>		

Ключевое слово `const` на указателях

`foo(&x)`

	<code>int x</code>	<code>int const x</code>
<code>void foo(int * p)</code>	✓	✗
<code>void foo(int const * p)</code>	✓	✓

Ключевое слово `const` на указателях

`foo(x)`

	<code>int x</code>	<code>int const x</code>
<code>void foo(int y)</code>		
<code>void foo(int const y)</code>		

Ключевое слово `const` на указателях

`foo(x)`

	<code>int x</code>	<code>int const x</code>
<code>void foo(int y)</code>	✓	✓
<code>void foo(int const y)</code>	✓	✓

Ключевое слово const

- `void foo(int p) { /* ... */ }`

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

Ключевое слово const

- `void foo(int p) { /* ... */ }`

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

- `p` — копия `x`
`const` оригинала и `const` копии никак не связаны

Ключевое слово const

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

```
const Foo f;  
f.field = 37;
```

Ключевое слово const

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

```
const Foo f;  
f.field = 37;
```

Ключевое слово const

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

```
const Foo f;  
f.field = 37;
```

Ключевое слово const

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

```
const Foo f;  
f.field = 37;
```

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 = 37;  
    }  
};
```

```
const Foo f;
```

Ключевое слово const

```
class Foo {  
public:  
    int field;  
    Foo() { ... }  
  
    void print() {  
        std::cout << this->field;  
    }  
};
```

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

Ключевое слово const

```
class Foo {  
public:  
    int field;  
    Foo() { ... }  
  
    void print() {  
        std::cout << this->field;  
    }  
};
```

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

Ключевое слово const

```
class Foo {  
public:  
    int field;  
    Foo() { ... }  
  
    void print(/* Foo * this */) {  
        std::cout << this->field;  
    }  
};
```

```
const Foo f;
```

```
f.print();
```


Ключевое слово const

```
class Foo {  
public:  
    int field;  
    Foo() { ... }  
  
    void print(/* Foo * this */) {  
        std::cout << this->field;  
    }  
};
```

```
const Foo f;
```

```
f.print();
```

- метод bar может нарушить свойство `const` объекта f

Ключевое слово const

```
class Foo {  
public:  
    int field;  
    Foo() { ... }  
  
    void print(/* Foo * this */) {  
        std::cout << this->field;  
        this->field = 42;  
    }  
};
```

```
const Foo f;
```

```
f.print();
```

- метод `bar` может нарушить свойство `const` объекта `f`

Ключевое слово const

```
class Foo {  
public:  
    int field;  
    Foo() { ... }  
  
    void print(/* Foo * this */) const {  
        std::cout << this->field;  
    }  
};
```

```
const Foo f;
```

```
f.print();
```

- метод bar утверждает, что не может нарушить свойство `const` объекта `f`

Ключевое слово const

```
class Foo {  
public:  
    int field;  
    Foo() { ... }  
  
    void print(/* Foo const * this */) const {  
        std::cout << this->field;  
    }  
};
```

```
const Foo f;
```

```
f.print();
```

- метод bar утверждает, что не может нарушить свойство `const` объекта `f`

Ключевое слово const

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

```
void print(/* Foo * this */) const {  
    std::cout << this->field;  
    this->field = 42;  
}
```

```
const Foo f;
```

```
f.print();
```

- метод bar утверждает, что не может нарушить свойство `const` объекта `f`
- попытка нарушить это утверждение — ошибка компиляции

const на полях

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

```
Foo f;
```

const на полях

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

```
Foo f;
```

- поле field объекта f не проинициализировано

const на полях

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

Foo f;

- поле field объекта f не проинициализировано
- **и его уже негде инициализировать!**

const на полях

```
class Foo {  
public:  
    const int field;  
    Foo() {  
        this->field = 42;  
    }  
};
```

Foo f;

- поле field объекта f не проинициализировано
- **и его уже нигде инициализировать!**

const на полях

```
class Foo {  
public:  
    const int field;  
    Foo() {  
        this->field = 42;  
    }  
};
```

Foo f;

- поле field объекта f не проинициализировано
- и его уже негде инициализировать!

const на полях

```
class Foo {  
public:  
    const int field;  
    Foo() : field(42) {  
  
    }  
};
```

```
Foo f;
```

- список инициализации спасает

const на полях

```
class Foo {  
public:  
    const int field = 42;  
    Foo() {  
  
    }  
};
```

```
Foo f;
```

- так тоже можно

Ключевое слово const

14 слайдов об одном ключевом слове

Но суть проста:

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

- Это указатели

Ссылки

- Это указатели
- почти

- 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)

- 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 языка не позволяет эти значения никаким образом получить

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

```
*ptr = 42;  
y = *ptr;
```

```
ptr = &y;
```

Ссылки

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

```
*ptr = 42;  
y = *ptr;
```

```
ptr = &y;
```

```
int x, y;  
int & ref = x; // инициализация ссылки
```

Ссылки

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

```
*ptr = 42;  
y = *ptr;
```

```
ptr = &y;
```

```
int x, y;  
int & ref = x; // инициализация ссылки
```

- неявное взятие адреса x



Ссылки

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

```
*ptr = 42;  
y = *ptr;
```

```
ptr = &y;
```

```
int x, y;  
int & ref = x; // инициализация ссылки
```

```
 r = 42; // запись по ссылке  
y =  r; // чтение по ссылке
```

- неявное взятие адреса `x`
- неявная операция разыменования

Ссылки

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

```
*ptr = 42;  
y = *ptr;
```

```
ptr = &y;
```

```
int x, y;  
int & ref = x; // инициализация ссылки
```

```
r = 42; // запись по ссылке  
y = r; // чтение по ссылке
```

```
r = y; // запись по ссылке
```

- неявное взятие адреса `x`
- неявная операция разыменования
- и отключить её нельзя

Ссылки

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

```
*ptr = 42;  
y = *ptr;
```

```
ptr = &y;
```

```
int x, y;  
int & ref = x; // инициализация ссылки
```

```
r = 42; // запись по ссылке  
y = r; // чтение по ссылке
```

```
r = y; // запись по ссылке  
      // меняет x
```

- неявное взятие адреса `x`
- неявная операция разыменования
- и отключить её 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);
```

Передача аргумента по ссылке

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

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

```
foo(5);  
// фактически попытка взять адрес числа 5
```

Передача аргумента по ссылке

```
void foo(int & r) {  
    r = 42;  
}
```

```
int x = 42;  
foo(x);  
assert(x == 37);
```

- передача по ссылке подобна передаче по указателю языка C

Передача аргумента по ссылке

```
class Foo {  
public:  
    int x;  
};  
  
void bar(Foo & f) {  
    f.x = 37;  
}
```

```
Foo obj;  
  
obj.x = 42;  
bar(obj);  
assert(obj.x == 37);
```

Передача аргумента по ссылке

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

```
void bar(Foo & f) {  
    f.x = 37;  
}
```

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

Передача аргумента по ссылке

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

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

```
void bar(Foo const & f) {  
    f.x = 37;  
}
```

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

- Для экономии действий (копируется адрес объекта, а не значение)
- Для изменения переданных параметров
- Изменяемость параметра контролируется спецификатором `const`
- В отличие от C, нельзя по виду вызова судить о неизменяемости переданных переменных, нужно смотреть описание метода

Конструктор копирования

```
class Vector {  
    int size;  
    double * data;  
public:  
    Vector(int size)  
        : size(size)  
        , data(new double[size]) {}  
  
    ~Vector() {  
        delete[] this->data;  
    }  
};
```

```
void foo(Vector b) {  
    /* ... */  
}  
  
Vector a(6);  
foo(a);
```

Конструктор копирования

```
class Vector {  
    int size;  
    double * data;  
public:  
    Vector(int size)  
        : size(size)  
        , data(new double[size]) {}  
  
    ~Vector() {  
        delete[] this->data;  
    }  
};
```

```
void foo(Vector b) {  
    /* ... */  
}
```

```
Vector a(6);  
foo(a);
```

- b — копия a
- копирование объектов, как и в языке Си, по умолчанию — побайтовое, без углубления по адресам

Конструктор копирования

a:

data →

size = 6

```
void foo(Vector b) {  
    /* ... */  
}
```

```
Vector a(6);  
foo(a);
```

- b — копия a
- копирование объектов, как и в языке Си, по умолчанию — побайтовое, без углубления по адресам

Конструктор копирования

a:



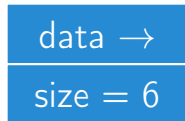
```
void foo(Vector b) {  
    /* ... */  
}
```

```
Vector a(6);  
foo(a);
```

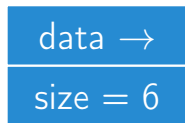
- b — копия a
- копирование объектов, как и в языке Си, по умолчанию — побайтовое, без углубления по адресам

Конструктор копирования

a:



b:



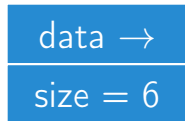
```
void foo(Vector b) {  
    /* ... */  
}
```

```
Vector a(6);  
foo(a);
```

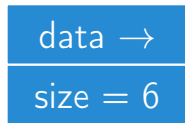
- b — копия a
- копирование объектов, как и в языке Си, по умолчанию — побайтовое, без углубления по адресам

Конструктор копирования

a:



b:



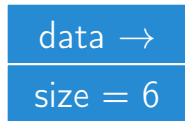
```
void foo(Vector b) {  
    /* ... */  
} // b.~Vector()
```

```
Vector a(6);  
foo(a);
```

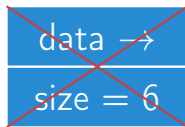
- b — копия a
- копирование объектов, как и в языке Си, по умолчанию — побайтовое, без углубления по адресам

Конструктор копирования

a:



b:



```
void foo(Vector b) {  
    /* ... */  
} // b.~Vector()
```

```
Vector a(6);  
foo(a);
```

- b — копия a
- копирование объектов, как и в языке Си, по умолчанию — побайтовое, без углубления по адресам

Конструктор копирования по умолчанию

- Побайтово, без углубления по адресам
- Подходит для простых “однослойных” структур, не владеющих памятью, на которую указывают

Конструктор копирования по умолчанию

Эти типы данных не владеют никакой памятью

```
class Point {  
    double x;  
    double y;  
    double z;  
public:  
    /* ... */  
};
```

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

Конструктор копирования по умолчанию

Для более сложных типов нужно определять свой способ копирования

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

Конструктор копирования

Вызывается в трёх случаях:

Конструктор копирования

Вызывается в трёх случаях:

- Создание нового объекта (явно)

```
Vector b(a);  
Vector c = a;
```

Конструктор копирования

Вызывается в трёх случаях:

- Создание нового объекта (явно)

```
Vector b(a);  
Vector c = a;
```

- Передача объекта в функцию по значению

```
void foo(Vector v) { /* ... */ }  
foo(a);
```

Конструктор копирования

Вызывается в трёх случаях:

- Создание нового объекта (явно)

```
Vector b(a);  
Vector c = a;
```

- Передача объекта в функцию по значению

```
void foo(Vector v) { /* ... */ }  
foo(a);
```

- Возврат объекта из функции по значению

```
Vector bar() {  
    Vector v(8);  
    /* ... */  
    return v;  
}
```

Конструктор копирования

a:

data →

size = 6

Конструктор копирования

a:

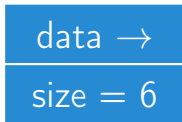


Конструктор копирования

a:



b:

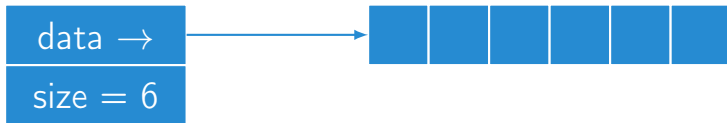


Конструктор копирования

a:



b:

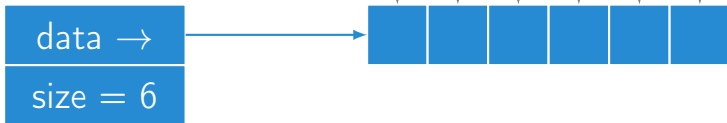


Конструктор копирования

a:



b:

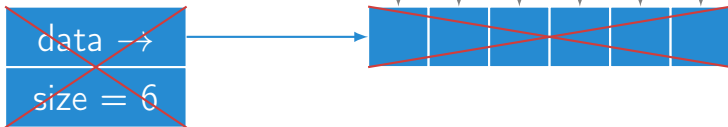


Конструктор копирования

a:



b:



Конструктор копирования

```
class Vector {  
public:  
    Vector(Vector const & that)  
        : size(that.size)  
        , data(new double[that.size])  
    {  
        for (int i = 0; i < size; ++i) {  
            data[i] = that.data[i];  
        }  
    }  
};
```

Конструктор копирования

```
class Vector {  
public:  
    Vector(Vector const & that)  
        : size(that.size)  
        , data(new double[that.size])  
    {  
        for (int i = 0; i < size; ++i) {  
            data[i] = that.data[i];  
        }  
    }  
};
```

- Конструктор копирования должен принимать свой параметр по ссылке.

Конструктор копирования

```
class Vector {  
public:  
    Vector(Vector const & that)  
        : size(that.size)  
        , data(new double[that.size])  
    {  
        for (int i = 0; i < size; ++i) {  
            data[i] = that.data[i];  
        }  
    }  
};
```

- Конструктор копирования должен принимать свой параметр по ссылке.
- `const` можно не указывать, но это неприятно ограничивает область применения, так что смысла нет. Исключение — классы, копирование которых невозможно без модификации оригинала.

О чём это мы?

Перегрузка операторов

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

Перегрузка операторов

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

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

Перегрузка операторов

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

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

- Для корректности — иначе для вызова конструктора копий придётся вызывать конструктор копий.

Перегрузка операторов

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

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

- Для эффективности — чтобы не вызывать конструктор копий лишний раз.

Перегрузка операторов

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

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

- Для расширения области применения;
- Для надёжности;

Перегрузка операторов

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

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

- Для расширения области применения;
- Для надёжности;
- b может быть константным

Перегрузка операторов

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

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

- Для расширения области применения;
- Для надёжности;

Перегрузка операторов

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

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

- Для расширения области применения;
- Для надёжности;
- а может быть константным

Перегрузка операторов

```
class Vector {  
    double * data;  
    int size;  
public:  
    Vector(Vector const & that);  
  
    Vector operator+ (Vector const & 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; // return Vector(result)  
    } // result.~Vector()  
};
```

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

- Для корректного возврата по значению;

Перегрузка операторов

```
class Vector {  
    double * data;  
    int size;  
public:  
    Vector(Vector const & that);  
  
    Vector & operator+ (Vector const & 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; // return Vector(result)  
    } // result.~Vector()  
};
```

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

- можно ли возвращать по ссылке и сэкономить на вызове конструктора копий?

Перегрузка операторов

```
class Vector {  
    double * data;  
    int size;  
public:  
    Vector(Vector const & that);  
  
    Vector & operator+ (Vector const & 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; // return Vector(result)  
    } // result.~Vector()  
};
```

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

- нельзя, ведь ссылка будет возвращена на объект, который удалится после выхода из функции.

Также, как и нельзя возвращать адрес локальной переменной.

Что можно ещё перегрузить?

- Все операторы, определённые в C++
- кроме `::` `.` `*` `?` `:` `sizeof` `typeid`
- При перегрузке `&&` и `||` теряется свойство ленивости

Оператор присваивания

- В отличие от конструктора копий работает с уже созданным объектом
- Должен сначала очистить старое состояние
- Потом скопировать оригинал

Оператор присваивания

- В отличие от конструктора копий работает с уже созданным объектом
- Должен сначала очистить старое состояние (деструктор)
- Потом скопировать оригинал (конструктор копий)

Оператор присваивания

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

Правило трёх

Если в классе есть один из:

- Деструктор
- Конструктор копирования
- Оператор присваивания

То в нём **должны** быть оставшиеся два

Не путать

```
Vector a(8);  
Vector b = a;    // конструктор копий  
Vector c(6);  
c = a;           // оператор присваивания
```

Перегрузка операторов

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

- упражнение #1 — определить назначение всех символов в этом тексте
- упражнение #2 — 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
- Правило трёх

Проверочные вопросы

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

Q & A