Перегрузка методов

> Методы классов - это просто функции, в которые неявно передается указатель на сам класс

```
class Connection
{
public:
    void send(int value);
    void send(const std::string& value);
};
```

> Конструкторы - это тоже функции и их тоже можно перегружать.

```
class Connection
{
public:
    Connection(const std::string& address, uint16_t port);
    Connection(const IpV4& address, uint16_t port);
    Connection(const IpV6& address, uint16_t port);
};
```

> Деструкторы - тоже функции, но перегружать нельзя.

Параметры по умолчанию

```
class Connection
{
public:
    Connection(const std::string& address, uint16_t port = 8080);
};
```

```
class Connection
{
public:
    Connection(const std::string& address = "localhost", uint16_t port = 8080);
};
```

> Пропусков в параметрах по умолчанию быть не должно, начинаться они могут не с первого аргумента, но заканчиваться должны на последнем.

Явные приведения типов

static_cast

Явное приведение встроенных типов:

```
double x = 1.5;

// Убираем предупреждение компилятора,
// четко показываем свои намерения
int y = static_cast<int>(x);
```

Приведение указателя на void

```
int* data = static_cast<int*>(malloc(100 * sizeof(int)));
```

Приведение вверх и вниз по иерархии

```
struct A {};
struct B : public A{};
struct C : public A{};
struct D {};
A* a = new B();
// Ошибка компиляции
D* d = static_cast<D*>(a);
// Безопасно
B* b = static_cast<B*>(a);
// Неопределенное поведение на
// этапе выполнения
C* c = static_cast<C*>(a);
```

const_cast

Снятие или добавление константности.

```
int x = 5;
const int* cpx = &x;
int* px = const_cast<int*>(cpx);
```

dynamic_cast

1. В базовом классе должна быть хотя бы одна виртуальная функция
2. Требуется RTTI (Runtime Type Information)

```
struct A {};
struct B : public A{};
struct C : public A{};
struct D {};
A* a = new B();
// Ok
B* b = dynamic_cast<B*>(a);
// nullptr
C* c = dynamic_cast<C*>(a);
// nullptr
D* d = dynamic_cast<D*>(a);
```

Если приводить не указатели, а ссылки, то в случае неудачного приведения будет выброшено исключение `std::bad_cast`

> dynamic_cast - признак плохого дизайна

> dynamic_cast - дорог

reinterpret_cast

Приведение одного типа к другому через указатель или ссылку не выполняя никаких проверок.

```
D* d = reinterpret_cast<D*>(a);
int x = reinterpret_cast<int>(a);
```

C-cast

```
B* b = (B*) a;
```

1. Компилятор попробует использовать static_cast
2. Если это не удалось, то reinterpret_cast
3. По необходимости будет добавлен const_cast

Неявные приведения типов

```
int x = 5;
double y = x;
```

```
struct A
{
    A(int x) {}
};

A a = 5;
```

```
struct A
{
    A(int x, int y = 3) {}
};

A a = 5;
```

```
class BigInt
{
public:
    BigInt(int64_t value) {}
};

BigInt x = 5;
```

```
struct A
{
    explicit A(int x) {}
};

A a = 5; // Ошибка
```

Операторы

Операторы сравнения

```
class BigInt
{
    static constexpr size_t Size = 256;
    uint8_t data_[Size];
public:
    bool operator==(const BigInt& other) const
    {
        if (this == &other)
            return true;

        for (size_t i = 0; i < Size; ++i)
            if (data_[i] != other.data_[i])
                return false;

        return true;
    }

    bool operator!=(const BigInt& other) const
    {
        return !(*this == other);
    }
};

BigInt x = 5;

if (c == 5)
    ...
```

Еще операторы сравнения:

- Меньше <
- Больше >
- Меньше или равно <=
- Больше или равно >=

Бинарные арифметические операторы

Попробуем написать метод осуществлющий сложение двух BigInt:

```
class BigInt
{
    const BigInt& operator+(BigInt& other)
    {
        ...
        return *this;
    }
};

BigInt x = 3;
BigInt y = 5;
BigInt z = x + y; // ок
BigInt z = x + y + x; // ошибка
```

```
// x + y -> const BigInt
tmp = x.operator+(y)
// tmp + z
tmp.operator+(x)
// opearator - не константный метод,
// а tmp - константный объект
```

```
class BigInt
{
    BigInt& operator+(BigInt& other)
    {
        ...
        return *this;
    }
};

BigInt x = 3;
BigInt y = 5;
BigInt z = x + y + x; // ок, но x изменился
```

```
class BigInt
{
    BigInt operator+(BigInt& other)
    {
        BigInt tmp;
        ...
        return tmp;
    }
};


BigInt x = 3;
BigInt y = 5;
BigInt z = x + y + x; // ок
```

```
BigInt x = 3;
const BigInt y = 5;
BigInt z = x + y + x; // передача константного
// объекта y по неконстантной ссылке
```

```
class BigInt
{
    BigInt operator+(const BigInt& other)
    {
        BigInt tmp;
        ...
        return tmp;
    }
};

BigInt x = 3;
```

```
const BigInt y = 5;
BigInt z = x + y + x; // ок
```

```
const BigInt x = 3;
const BigInt y = 5;
BigInt z = x + y + x; // передача константного
// объекта x по неконстантной ссылке
```

Финальный вариант:

```
class BigInt
{
    BigInt operator+(const BigInt& other) const
    {
        BigInt tmp;
        ...
        return tmp;
    }
};
```

Операторы могут не быть членами класса:

```
class Int128 {};
class BigInt
{
    BigInt operator+(const Int128& other) const
    {
        ...
    }
};
BigInt x = 3;
Int128 y = 5;
BigInt z = x + y; // ok
BigInt z = y + x; // у Int128 нет оператора
// сложения с BigInt
```

```
class BigInt
{
    friend BigInt operator+(const Int128& x, const BigInt& y);
};

BigInt operator+(const Int128& x, const BigInt& y)
{
    ...
}
```

Еще операторы:

- Вычитание -
- Деление /
- Умножение *
- Остаток от деления %

> Для операторов действует стандартный приоритет арифметических операторов

Унарные арифметические операторы

```
BigInt x = 3;
BigInt y = -x;

class BigInt
{
    bool isNegative_;
public:
    BigInt operator-() const
    {
        BigInt tmp(*this);
```

```
        tmp.isNegative_ = !isNegative_;
        return tmp;
    }
};
```

Для симметрии есть унарный плюс.

Операторы инкремента

```
Int x = 3;
++x;
x++;

class BigInt
{
    void increment()
    {
        ...
    }
public:
    // ++x
    BigInt& operator++()
    {
        increment();
        return *this;
    }
    // x++
    BigInt operator++(int)
    {
        BigInt tmp(*this);
        increment();
        return tmp;
    }
};
```

Операторы декремента аналогичны операторам инкремента.

Логические операторы

- Отрицание ! (унарный)
- И (логическое умножение) &&
- ИЛИ (логическое сложение) ||

Битовые операторы

- Инверсия ~
- И &
- ИЛИ |
- Исключающее ИЛИ (xor) ^
- Сдвиг влево <<
- Сдвиг вправо >>

Составное присваивание

Все арифметические, логические и побитовые операции только изменяющиие состояние объекта (с = в начале).

```
x += 3;
x *= 4;
```

```
class BigInt
{
    // не константная, так как объект изменяется
    const BigInt& operator+=(const BigInt& other)
    {
        ...
        return *this;
    }
};
```

```
BigInt x = 3;
(x += 5) + 7;

class BigInt
{
    BigInt operator+=(const BigInt& other)
    {
        BigInt tmp;
        ...
        return tmp;
    }
};
```

Оператор вывода в поток

Не метод класса.

```
std::ostream& operator<<(std::ostream& out, const BigInt& value)
{
    out << ...;
    return out;
}

BigInt x = 5;
std::cout << x;
```

Операторы доступа

Семантика доступа к массиву.

```
class Array
{
    uint8_t* data_;
public:
    const uint8_t& operator[](size_t i) const
    {
        return data_[i];
    }

    uint8_t& operator[](size_t i)
    {
        return data_[i];
    }
};

Array a;
a[3] = 4;

const Array b;
b[5] = 6; // Ошибка
auto x = b[1]; // Ok
```

Семантика указателя

```
class MyObject
{
public:
    void foo() {}
};

class MyObjectPtr
{
    MyObject* ptr_;
public:
    MyObjectPtr()
        : ptr_(new MyObject())
    {
    }

    ~MyObjectPtr()
    {
```

```
        delete ptr_;
    }

    MyObject& operator*()
    {
        return *ptr_;
    }

    const MyObject& operator*() const
    {
        return *ptr_;
    }

    MyObject* operator->()
    {
        return ptr_;
    }

    const MyObject* operator->() const
    {
        return ptr_;
    }
};

MyObjectPtr p;
p->foo();
(*p).foo();
```

Функтор

Позволяет работать с объектом как с функцией.

```
class Less
{
public:
    bool operator()(
        const BigInt& left, const BigInt& right) const
    {
        return left < right;
    }
};

Less less;
if (less(3, 5))
    ...
```

Другие операторы

- new
- delete
- ,

Сокрытие

```
struct A
{
    void foo() {} // 1
};

struct B
        : public A
{
    void foo() {} // 2
};

A a;
a.foo(); // Будет вызвана 1

B b;
b.foo(); // Будет вызвана 2
```

```
A* c = new B();
c->foo(); // Будет вызвана 1
```

Виртуальные функции

```
struct A
{
    virtual void foo() const {} // 1
};

struct B
    : public A
{
    void foo() const override {} // 2
};

A a;
a.foo(); // Будет вызвана 1

B b;
b.foo(); // Будет вызвана 2

A* c = new B();
c->foo(); // Будет вызвана 2

const A& d = B();
d.foo(); // Будет вызвана 2
```

В первых двух случаях используется раннее (статическое) связывание, еще на этапе компиляции компилятор знает какой метод вызвать.

В третьем случае используется позднее (динамическое) связывание, компилятор на этапе компиляции не знает какой метод вызвать, выбор нужного метода будет сделан во время выполнения.

Виртуальные функции в C

```
#include <stdio.h>

struct Device
{
    virtual void write(const char* message) {}
};

class Console : public Device
{
    int id_;
public:
    Console(int id)
        : id_(id)
    {
    }

    void write(const char* message) override
    {
        printf("Console %d: %s\n", id_, message);
    }
};

class Socket : public Device
{
    const char* address_;
public:
    Socket(const char* address)
        : address_(address)
    {
    }

    void write(const char* message) override
    {
        printf("Send %s to %s\n", message, address_);
    }
};
```

```
int main()
{
    Device* devices[] = {
        new Console(10),
        new Socket("10.0.0.1") };

    Device* dev1 = devices[0];
    dev1->write("A");

    Device* dev2 = devices[1];
    dev2->write("B");

    return 0;
}
```

```
Console 10: A
Send B to 10.0.0.1
```

```
#include <stdio.h>
#include <stdlib.h>

struct Device;

struct DeviceVirtualFunctionTable
{
    void (*write)(Device* self, const char* message);
};

struct Device
{
    DeviceVirtualFunctionTable vft_;
};

void Device_write(Device* self, const char* message)
{
    self->vft_.write(self, message);
}

struct Console
{
    DeviceVirtualFunctionTable vft_;
    int id_;
};

void Console_write(Device* self, const char* message)
{
    Console* console = (Console*) self;
    printf("Console %d: %s\n", console->id_, message);
}

Device* Console_new(int id)
{
    Console* instance = (Console*) malloc(sizeof(Console));
    instance->vft_.write = Console_write;
    instance->id_ = id;
    return (Device*) instance;
}

struct Socket
{
    DeviceVirtualFunctionTable vft_;
    const char* address_;
};

void Socket_write(Device* self, const char* message)
{
    Socket* socket = (Socket*) self;
    printf("Send %s to %s\n", message, socket->address_);
}

Device* Socket_new(const char* address)
{
    Socket* instance = (Socket*) malloc(sizeof(Socket));
```

```
    instance->vft_.write = Socket_write;
    instance->address_ = address;
    return (Device*) instance;
}

int main()
{
    Device* devices[] = {
        Console_new(10),
        Socket_new("10.0.0.1") };

    Device* dev1 = devices[0];
    Device_write(dev1, "A");

    Device* dev2 = devices[1];
    Device_write(dev2, "B");

    return 0;
}
```

```
Console 10: A
Send B to 10.0.0.1
```

Таблица виртуальных функций

Если в классе или в каком-либо его базовом классе есть виртуальная функция, то каждый объект хранит указатель на таблицу виртуальных функций.

Таблица представляет собой массив из указателей на функции.

```
struct A
{
    void foo() {}
    int x;
};

struct B
{
    virtual void foo() {}
    int x;
};

std::cout << sizeof(A) << '\n';
std::cout << sizeof(B) << '\n';
```

```
4
16
```

Виртуальный деструктор

```
struct A
{
    ~A()
    {
        std::cout << "A";
    }
};

struct B
    : public A
{
    ~B()
    {
        std::cout << "B";
        delete object_;
    }

    SomeObject* object_;
};
```

```
A* a = new B();
delete a;
```

```
A
```

> Произошла утечка, так как не был вызван деструктор, в котором мы освобождали ресурс.

```
struct A
{
    virtual ~A()
    {
    }
};
```

> Используете наследование? Сделайте деструктор виртуальным.

Чисто виртуальные функции (pure virtual)

```
class Device
{
public:
    virtual void ~Device() {}

    virtual void write(const char* message) = 0;
};

class ConsoleWriter
        : public Writer
{
public:
    void write(const char* message) override
    {
        std::cout << message;
    }
}
```

Абстрактные классы

Классы имеющие хоть одну чисто виртуальную функцию - абстрактные. При попытке создать их компилятор выдаст ошибку. Если в производном классе не сделать реализацию чисто виртуальной функции, то он тоже становится абстрактным.

> Абстрактные классы в C++ - продвинутые интерфейсные классы в других языках.

Виртуальные функции и параметры по умолчанию

```
struct A
{
    virtual void foo(int i = 10)
    {
        std::cout << i; // 1
    }
};

struct B
    : public A
{
    virtual void foo(int i = 20)
    {
        std::cout << i; // 2
    }
};

A* a = new B();
a->foo(); // Будет вызвана 2, вывод 10

B* b = new B();
b->foo(); // Будет вызвана 2, вывод 20
```

```
A* a = new A();
a->foo(); // Будет вызвана 1, вывод 10
```

> Лучше избегать параметров по умолчанию для виртуальных функций

Модификаторы доступа при наследовании

```
class A
{
public:
    int x_;
protected:
    int y_;
private:
    int z_;
};
```

Псевдокод! Поля базового класса после наследования имеют такие модификаторы:

```
class B : public A
{
public:
    int x_;
protected:
    int y_;
};

A* a = new B(); // Ok
```

```
class B : protected A
{
protected:
    int x_;
    int y_;
};

A* a = new B(); // Ошибка
```

```
class  B : private A
{
private:
    int x_;
    int y_;
};

A* a = new B(); // Ошибка
```

public - классическое ООП наследование

```
class Device
{
};

class NetworkAdapter
    : public Device
{
};

class DeviceManager
{
    void addDevice(Device* dev)
    {
    }
}

devManager.addDevice(new NetworkAdapter());
```

private - наследование реализации

```
class NetworkAdapter
    : public Device
    , private Loggable
{
};

Loggable* l = new NetworkAdapter(); // Ошибка
```

final

```
struct A final
{
};

struct B : public A // Ошибка
```

Множественное наследование

```
struct A
{
    virtual ~A() {}
    double x;
    double y;
};

struct B : public A { };

struct C : public A { };

struct D
    : public B
    , public C
{
};
```

```
+------+ +------+
|  A   | |  A   |
| x, y | | x, y |
| vtab | | vtab |
+------+ +------+
    ^       ^
    |       |
+------+ +------+
|  B   | |  C   |
+------+ +------+
     ^    ^
     |    |
    +------+
    |  D   |
    +------+
```

```
// 2 * 8(double) + 1 * 8(vtable)
sizeof(A) == 24
sizeof(B) == 24
sizeof(C) == 24
// 2 * 8(double) + 2 * 8(vtable)
sizeof(D) == 48
```

```
[A][B][D]
         [A][C]
```

```
struct A
{
    A(double x)
        : x(x)
        , y(0)
```

```
    {
    }
    virtual ~A() {}
    double x;
    double y;
};

struct B : public A
{
    B(double x)
        : A(x)
    {
        y = x * 2;
    }
};

struct C : public A
{
    C(double x)
        : A(x)
    {
        y = x * 2;
    }
};

struct D
    : public B
    , public C
{
    D()
        : B(2)
        , C(3)
    {
        B::y = 1;
        C::y = 2;
    }
};
```

Ромбовидное наследование

```
struct A
{
    virtual ~A() {}
    double x;
    double y;
};

struct B : virtual public A { };

struct C : virtual public A { };

struct D
    : public B
    , public C
{
};
```

```
// 2 * 8(double) + 1 * 8(vtable)
sizeof(A) == 24
// 2 * 8(double) + 2 * 8(vtable)
sizeof(B) == 32
sizeof(C) == 32
// 2 * 8(double) + 3 * 8(vtable)
sizeof(D) == 40
```

```
[B][D]
      [C]
         [A]
```

Вложенные классы

```
class Vector
{
public:
    class Iterator
    {
    };

private:
    char* data_;
};

Vector::Iterator it = ...
```

> Имеют доступ к закрытой части внешнего класса

Практическая часть

Нужно написать класс-матрицу, тип элементов int. В конструкторе задается количество рядов и строк. Поддерживаются операции: получить количество строк(rows)/столбцов(columns), получить конкретный элемент, умножить на число(*=), сравнение на равенство/неравенство. В случае ошибки выхода за границы бросать исключение:

```
throw std::out_of_range("")
```

Пример:

```
const size_t rows = 5;
const size_t cols = 3;

Matrix m(rows, cols);

assert(m.getRows() == 5);
assert(m.getColumns() == 3);

m[1][2] = 5; // строка 1, колонка 2
double x = m[4][1];

m *= 3; // умножение на число

Matrix m1(rows, cols);

if (m1 == m)
{
}
```

Для проверки на сайте класс должен быть оформлен как заголовочный файл, название класса - Matrix. Нужно скомпилировать файл `test.cpp`, в нем включается заголовочный файл `matrix.h` и тестируется. В случае успеха в выводе кроме строки done ничего больше не будет.

Подсказка

Чтобы реализовать семантику [][] понадобится прокси-класс. Оператор матрицы возвращает другой класс, в котором тоже используется оператор [] и уже этот класс возвращает значение.