Лекция 6. Наследование. Перегрузка операторов

Илья Макаров

ИТМО JB

12 октября 2021

Санкт-Петербург

Еще раз про множественное наследование

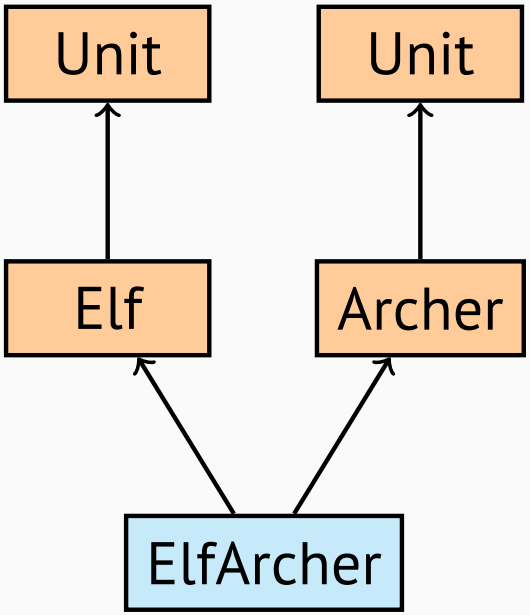
Множественное наследование (multiple inheritance) — возможность наследовать сразу несколько классов.

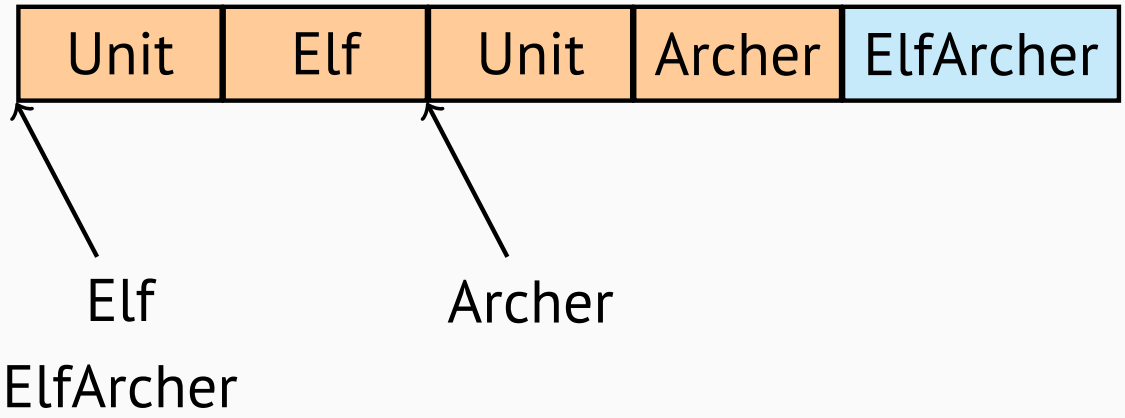
```
struct Unit {
    Unit(unitid id, int hp): id_(id), hp_(hp) {}
    virtual unitid id() const { return id_; }
    virtual int    hp() const { return hp_; }
private:
    unitid id_;
    int    hp_;
};
```

Еще раз про множественное наследование

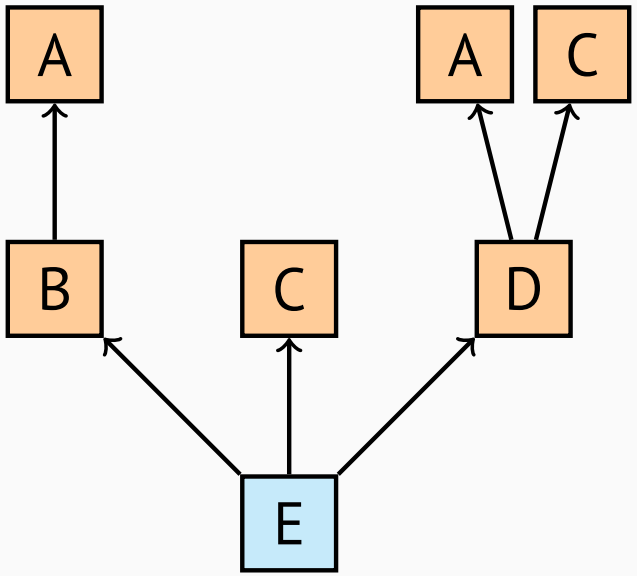
```
struct Elf:    Unit { ... };
struct Archer: Unit { ... };

struct ElfArcher: Elf, Archer {
    unitid id() const { return Elf::id(); }
    int    hp() const { return Elf::hp(); }
};
```

Представление в памяти





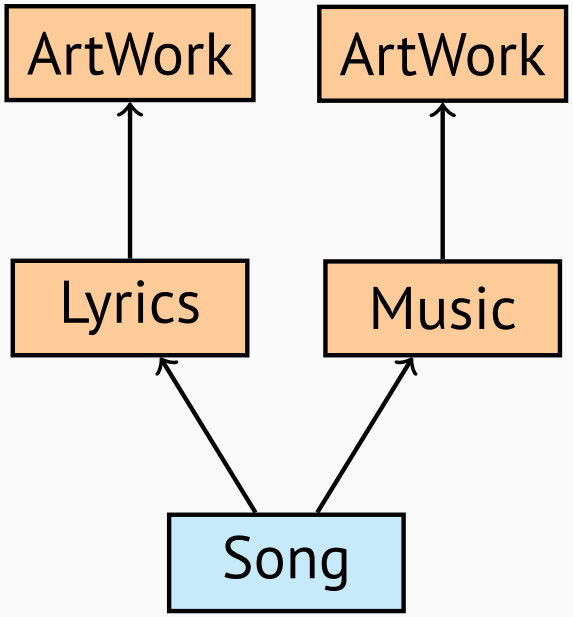
Создание и удаление объекта

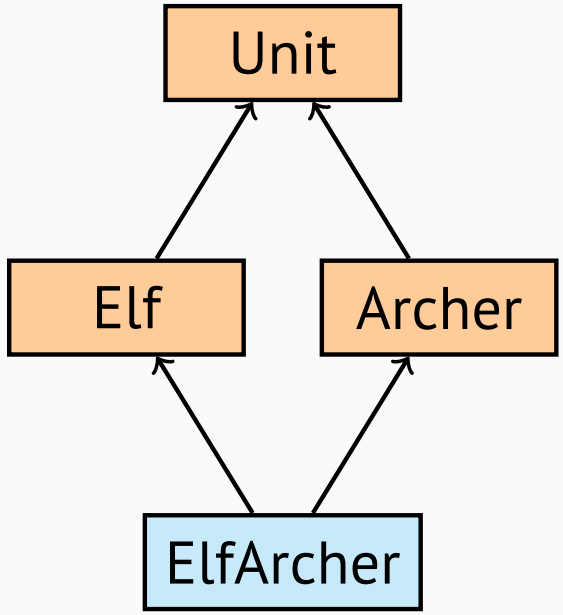


Порядок вызова конструкторов: `A`, `B`, `C`, `A`, `C`, `D`, `E`.
Деструкторы вызываются в обратном порядке.
Проблемы:

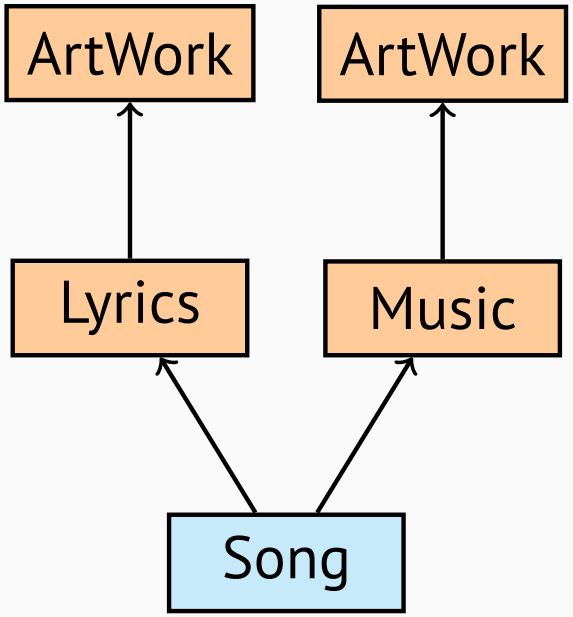
1. Дублирование A и C.
2. Недоступность первого C.

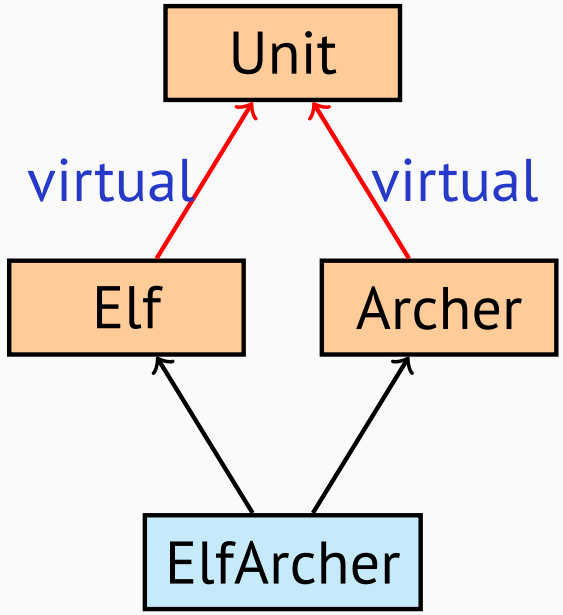
Виртуальное наследование





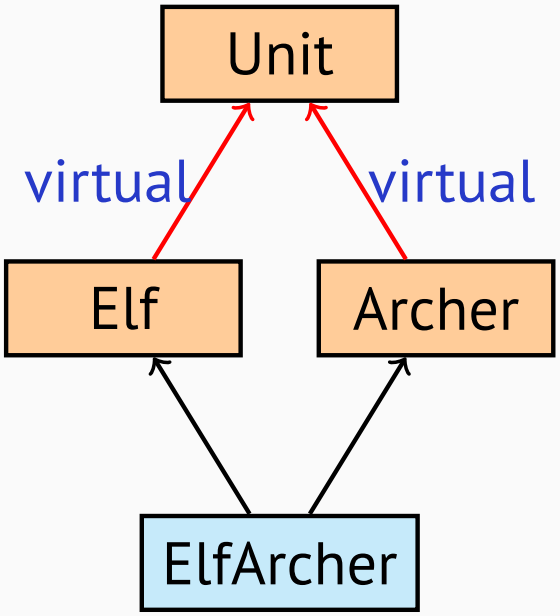
Виртуальное наследование

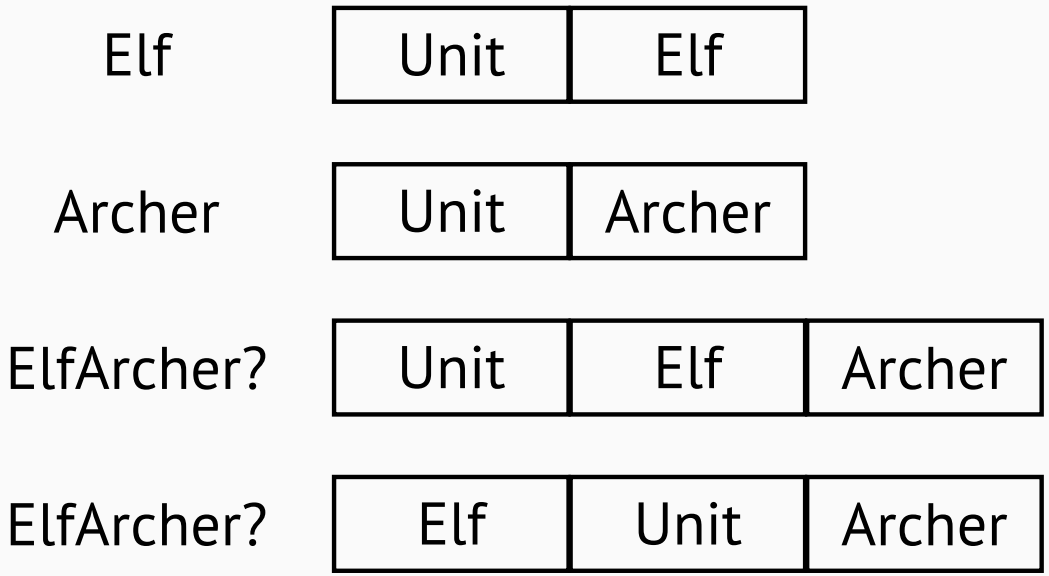




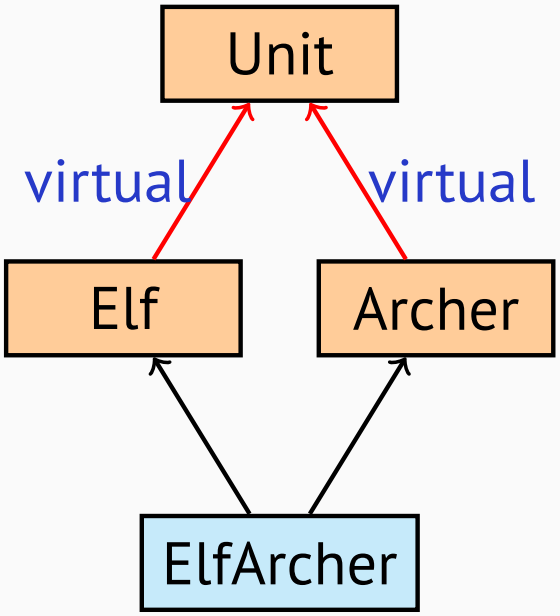

```
struct Unit {};
struct Elf:  virtual Unit {};
struct Archer: virtual Unit {};
struct ElfArcher: Elf, Archer {};
```

Как устроено расположение в памяти?

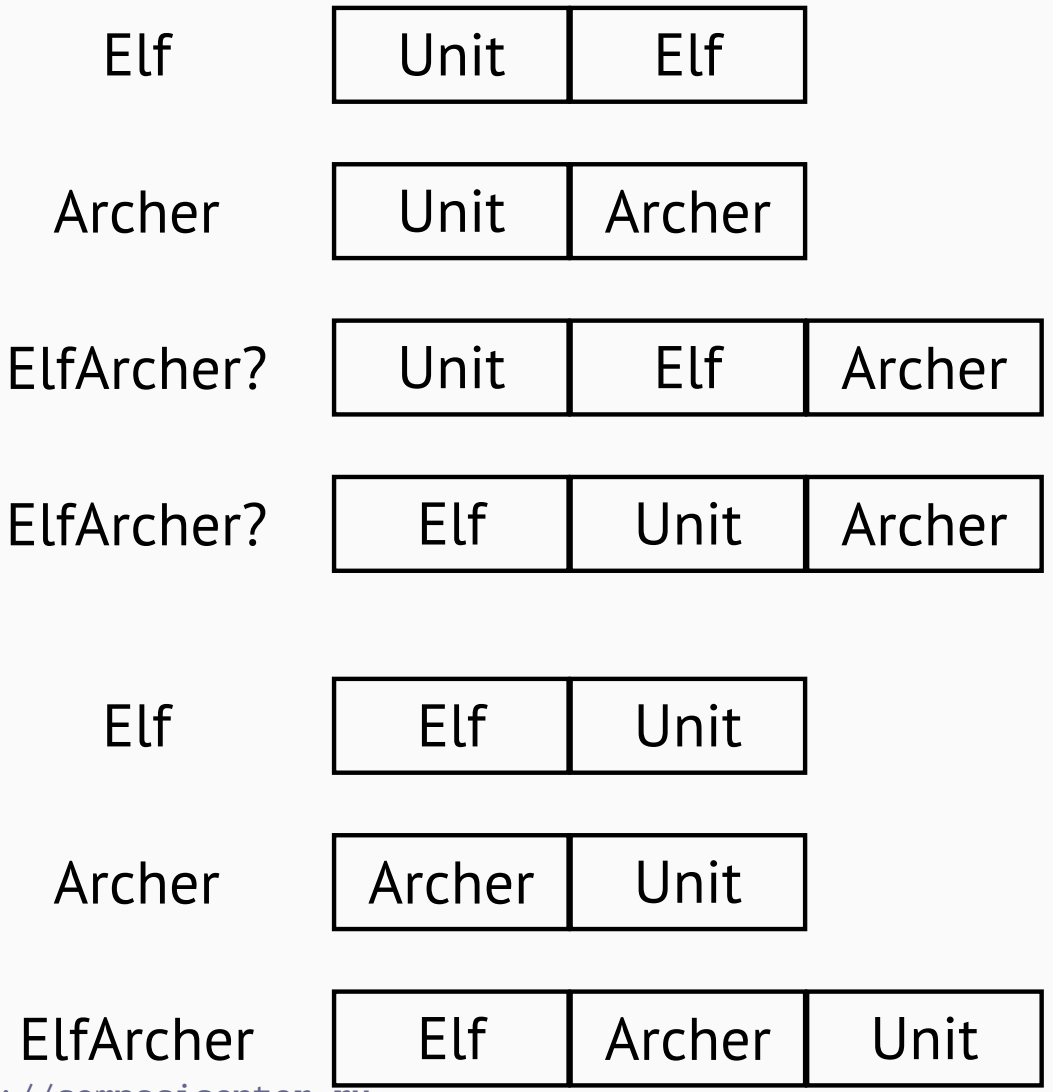




Как устроено расположение в памяти?



На самом деле.



Доступ через таблицу виртуальных методов

```
struct Unit {
    unitid id;
};
struct Elf   : virtual Unit { };
struct Archer : virtual Unit { };
struct ElfArcher : Elf, Archer { };
```

Рассмотрим такой код:

```
    Elf * e = (rand() % 2)? new Elf() : new ElfArcher();
    unitid id = e->id; // (*)
```

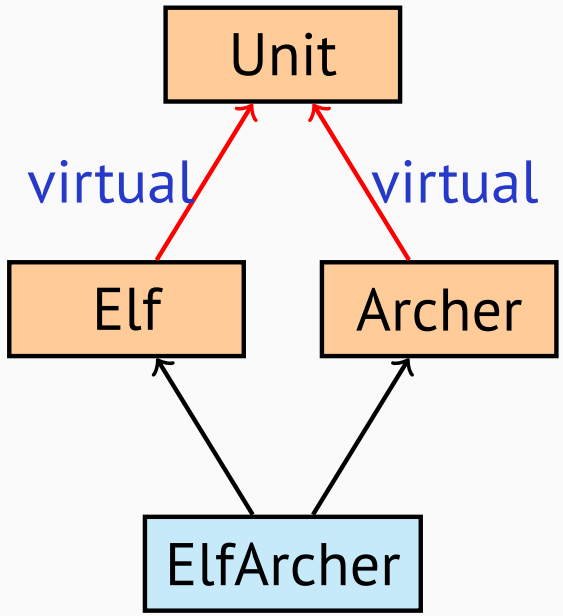
Строка (*) будет преобразована в строку

```
    unitid id = e->__getUnitPtr__()->id;
```

где `__getUnitPtr__()` – это служебный виртуальный метод.

Кто вызывает конструктор базового класса?

```
struct Unit {
    Unit() = default;
    Unit(unitid id, int health);
};
struct Elf: virtual Unit {
    explicit Elf(unitid id)
        : Unit(id, 100) {}
};
struct Archer: virtual Unit {
    explicit Archer(unitid id)
        : Unit(id, 120) {}
};
struct ElfArcher: Elf, Archer {
    explicit ElfArcher(unitid id)
        : Elf(id)
        , Archer(id) {}
};
```



Кто вызывает конструктор базового класса?

```
struct Unit {
    // Unit() = default;
    Unit(unitid id, int health);
};
struct Elf: virtual Unit {
    explicit Elf(unitid id)
        : Unit(id, 100) {}
};
struct Archer: virtual Unit {
    explicit Archer(unitid id)
        : Unit(id, 120) {}
};
struct ElfArcher: Elf, Archer {
    explicit ElfArcher(unitid id)
        : Elf(id)
        , Archer(id) {}
};
```

Кто вызывает конструктор базового класса?

```
struct Unit {
    Unit(unitid id, int health);
};
struct Elf: virtual Unit {
    explicit Elf(unitid id)
        : Unit(id, 100) {}
};
struct Archer: virtual Unit {
    explicit Archer(unitid id)
        : Unit(id, 120) {}
};
struct ElfArcher: Elf, Archer {
    explicit ElfArcher(unitid id)
        : Unit(id, 150)
        , Elf(id)
        , Archer(id) {}
};
```

Заключение

- Не используйте множественное наследование для наследования реализации.
- Используйте концепцию интерфейсов (классы без реализаций и членов данных).
- Помните о неприятностях, связанных с множественным наследованием.
- Хорошо подумайте перед тем, как использовать виртуальное наследование.
- Помните о неприятностях, связанных с виртуальным наследованием.

Основные операторы

Арифметические

- Унарные: префиксные `+ - ++ --`, постфиксные `++ --`
- Бинарные: `+ - * / % += -= *= /= %=`

Битовые

- Унарные: `~`.
- Бинарные: `& | ^ &= |= ^= >> << >>= <<=`.

Логические

- Унарные: `!`.
- Бинарные: `&& ||`.
- Сравнения: `== != > < >= <=`
- C++20: `<=>`

Другие операторы

1. Оператор присваивания: `=`
2. Специальные:
   - префиксные `* &`,
   - постфиксные `-> ->*`,
   - особые `, . .* ::`
3. Скобки: `[] ()`
4. Оператор приведения `(type)`
5. Тернарный оператор: `x ? y : z`
6. Работа с памятью: `new new[] delete delete[]`

Нельзя перегружать операторы `. ::` и тернарный оператор.

Перегрузка операторов

```
Vector operator-(Vector const& v) {
    return Vector(-v.x, -v.y)
}

Vector operator+(Vector const& v,
                 Vector const& w) {
    return Vector(v.x + w.x, v.y + w.y);
}

Vector operator*(Vector const& v, double d) {
    return Vector(v.x * d, v.y * d);
}

Vector operator*(double d, Vector const& v) {
    return v * d;
}
```

Перегрузка операторов внутри классов

NB: Обязательно для `(type) [] () -> ->* =`

```
struct Vector {
    Vector operator-() const { return Vector(-x, -y); }
    Vector operator-(Vector const& p) const {
        return Vector(x - p.x, y - p.y);
    }
    Vector & operator*=(double d) {
        x *= d;
        y *= d;
        return *this;
    }
    double operator[](size_t i) const {
        return (i == 0) ? x : y;
    }
    bool    operator()(double d)   const   { ... }
    void    operator()(double a, double b) { ... }
    double x, y;
};
```

Перегрузка инкремента и декремента

```
struct BigNum {
    BigNum & operator++() { //prefix
        //increment
        ...
        return *this;
    }

    BigNum operator++(int) { //postfix
        BigNum tmp(*this);
        ++(*this);
        return tmp;
    }
    ...
};
```

Переопределение операторов ввода-вывода

```
#include <iostream>

struct Vector { ... };

std::istream& operator>>(std::istream & is,
                         Vector & p) {
    is >> p.x >> p.y;
    return is;
}

std::ostream& operator<<(std::ostream &os,
                         Vector const& p) {
    os << p.x << ' ' <<  p.y;
    return os;
}
```

Умный указатель

Реализует принцип: *“Получение ресурса есть инициализация”*
Resource Acquisition Is Initialization (RAII)

```
struct SmartPtr {
   Data & operator*()  const {return *data_;}
   Data * operator->() const {return  data_;}
   Data * get()        const {return  data_;}
   ...
private:
   Data * data_;
};

bool operator==(SmartPtr const& p1,
                SmartPtr const& p2) {
    return p1.get() == p2.get();
}
```

Умный указатель

```
struct Data { int id; };

struct SmartPtr {
    Data & operator*()  const {return *data_;}
    Data * operator->() const {return  data_;}
    Data * get()        const {return  data_;}
    ...
private:
    Data * data_;
};

SmartPtr p;
p->id; // p.operator->()->id;
p->foo() // error
```

Указатели на функции

Кроме указателей на значения в C++ присутствуют три особенных типа указателей:

1. указатели на функции (унаследованно из C),
2. указатели на методы,
3. указатели на поля классов.

Указатели на функции (и методы) используются для

1. параметризация алгоритмов,
2. обратных вызовов (callback),
3. подписки на события (шаблон Listener),
4. создание очередей событий/заданий.

Указатели на функции: параметризация алгоритмов

```
bool less(double a, double b) { return a < b; }

void sort(double * p, double * q,
          bool (*cmp)(double, double)) {
    for (double * m = p; m != q; ++m)
        for (double * r = m; r + 1 != q; ++r)
            if ( cmp(*(r + 1), *r) )
                swap(*r, *(r + 1));
}
int main() {
    double m[100];
    sort(m, m + 100, &less);
}
```

Сразу о полезности using

Что здесь объявлено?

```
char * (*func(int, int))(int, int, int *, float);
```

Функция двух целочисленных параметров, возвращающая указатель на функцию, которая возвращает указатель на char и имеет собственный список формальных параметров вида: (int, int, int *, float)

Как стоило это написать:

```
using SomeFunction
    = std::function<char*(int, int, int *, float)>;
SomeFunction func(int, int);
```

Указатели на методы

В отличие от указателей на функции требуют экземпляр класса.

```
struct Person {
    string name() const;
    string surname() const;
};
using MPTR = string (Person::*)() const;

void print(Person const& p) {
    static MPTR im[2] = {
        &Person::name,
        &Person::surname};

    for (size_t i = 0; i != 3; ++i)
        cout << (p.*im[i])(); // operator .*
}
```

Указатели на члены данных

Похожи на указатели на методы.

```
struct Person {
    string name;
    string surname;
};

using DPTR = string Person::*;

void print(Person const& p) {
    static DPTR im[2] = {
        &Person::name,
        &Person::surname};

    for (size_t i = 0; i != 3; ++i)
        cout << (p.*im[i]); // operator .*
}
```

Резюме по синтаксису

Указатели на методы и поля класса.

```
struct Student {
    string name () const { return name_; }
    string name_;
};
int main() {
    string (Student::*mptr)() const = &Student::name;
    string  Student::*dptr          = &Student::name_;
    Student s;
    Student * p = &s;
    (s.*mptr)() == (p->*mptr)();
    (s.*dptr)   == (p->*dptr);
}
```

Резюме по синтаксису

```
struct Foo {
    int i;
    void f();
};

int main () {
    Foo foo;
    Foo* fooPtr = &foo;
    int Foo::* iPtr = &Foo::i;
    void (Foo::*memFuncPtr)() = &Foo::f;

    foo.*iPtr = 0;
    fooPtr->*iPtr = 0;
    (foo.*memFuncPtr)();
    (fooPtr->*memFuncPtr)();
}
```

Резюме по синтаксису

Начиная с C++17 для вызова функций по указателю удобно использовать `std::invoke`.

```
int main() {
    Foo foo;
    Foo* fooPtr = &foo;
    auto iPtr = &Foo::i;
    auto memFuncPtr = &Foo::f;

    std::invoke(iPtr, foo) = 0;
    std::invoke(iPtr, fooPtr) = 0;

    std::invoke(memFuncPtr, foo);
    std::invoke(memFuncPtr, fooPtr);
}
```

Оператор ->*

```
template<typename ItemType>
struct List {
    List(ItemType *head,
         ItemType * ItemType::*nextMemPointer)
    : m_head(head)
    , m_nextMemPointer(nextMemPointer) { }

    void addHead(ItemType *item) {
    (item ->* m_nextMemPointer) = m_head;
    m_head = item;
    }

private:
    ItemType *m_head;
    ItemType * ItemType::*m_nextMemPointer;
};
```

Оператор приведения

```
struct String {
    operator bool() const {
        return size_ != 0;
    }

    operator char const *() const {
        if (*this)
            return data_;
        return "";
    }

private:
    char * data_;
    size_t size_;
};
```

Операторы с особым порядком вычисления

```
int main() {
    int a = 0;
    int b = 5;
    (a != 0) && (b = b / a);
    (a == 0) || (b = b / a);

    foo() && bar();
    foo() || bar();
    foo(), bar();
}
```

Операторы с особым порядком вычисления

```
struct P
{
    int data = 0;
};

// no lazy semantics
Tribool operator&&(Tribool const& b1,
                   Tribool const& b2);

int main() {
    P * p = new P();
    // lazy semantics
    if (p && p->data)
    {
        ...
    }
}
```

Переопределение арифметических операторов

```
struct String {
    String( char const * cstr ) { ... }
    String & operator+=(String const& s) {
        ...
        return *this;
    }
    //String operator+(String const& s2) const {...}
};
String operator+(String s1, String const& s2) {
    return s1 += s2;
}
```

```
    String s1("world");
    String s2 = "Hello " + s1;
```

“Правильное” переопределение операторов сравнения

```
bool operator==(String const& a, String const& b)
{ return ...  }
bool operator!=(String const& a, String const& b)
{ return !(a == b); }

bool operator<(String const& a, String const& b)
{ return ...  }
bool operator>(String const& a, String const& b)
{ return b < a; }
bool operator<=(String const& a, String const& b)
{ return !(b < a); }
bool operator>=(String const& a, String const& b)
{ return !(a < b); }
```

C++20 переопределение операторов сравнения

```
struct Record
{
    std::string name;
    unsigned int floor;
    double weight;
    auto operator<=>(const Record&) const = default;
};

// now can be compared with ==, !=, <, <=, >, and >=
```

C++20 переопределение операторов сравнения

```
struct Base {
    std::string zip;
    auto operator<=>(const Base&) const = default;
};
struct TotallyOrdered : Base {
    std::string tax_id;
    std::string first_name;
    std::string last_name;
public:
    // custom operator<=> because we want to compare last
names first
    std::strong_ordering operator<=>(
        const TotallyOrdered& that) const;
};
```

C++20 переопределение операторов сравнения

```
std::strong_ordering TotallyOrdered::operator<=>(
    const TotallyOrdered& that) const {
    if (auto cmp = (Base&)(*this) <=> (Base&)that; cmp != 0)
        return cmp;
    if (auto cmp = last_name <=> that.last_name; cmp != 0)
        return cmp;
    if (auto cmp = first_name <=> that.first_name; cmp != 0)
        return cmp;
    return tax_id <=> that.tax_id;
}
```

О чём стоит помнить

- Стандартная семантика операторов.

```
void operator+(A const & a, A const& b) {}
```

- Приоритет операторов.

```
    Vector a, b, c;
    c = a + a ^ b * a; //?????
```

- Хотя бы один из параметров должен быть пользовательским.

```
    void operator*(double d, int i) {}
```