Объектно-ориентированное программирование

Object-oriented programming

VIII. Полиморфизм

Polymorphism

Что такое OOP?

https://youtube.com/clip/UgkxMjTVfez-AB1gsyvoIpeprL2pc5YxvGjB?si=VIhZaa0uoTkcqI6n

"I think that object orientedness is almost as much of a **hoax** as artificial intelligence."

A. Stepanov

"The evolution of languages from **untyped** universes to **monomorphic** and then **polymorphic** type systems[...], mechanisms for polymorphism such as **overloading**, **coercion**, **subtyping**, and **parameterization**[...], a unifying framework for polymorphic type systems[...] in terms of the typed λ-calculus augmented to include **binding of types by quantification** as well as **binding of values by abstraction**."

L. Cardelli

"On Understanding Types, Data Abstraction, and Polymorphism", 1985

http://lucacardelli.name/Papers/OnUnderstanding.A4.pdf

Приведение типов и перегрузка

```
3   + 4
3.0 + 4
3   + 4.0
3.0 + 4.0
```

Приведение типов и перегрузка

```
3     +   4
3.0   +   4
3     +   4.0
3.0   +   4.0
```

- оператор “+” перегружен четыре раза
- оператор “+” перегружен два раза: для целых и дробных чисел
- оператор “+” не перегружен, а определен только для дробных чисел

Приведение типов и перегрузка

3 + 4
4 + 3
“3” + “4”
“4” + “3”

leon.brinzan@iis.utm.md

Приведение типов и перегрузка

```
3     +   4    // 7
4     +   3    // 7
“3”   +   “4”  // “34”
“4”   +   “3”  // “43”
```

Перегружать операторы так, как оператор **+** перегружен для строк в **C++**, в подавляющем большинстве случаев не следует. Такая перегрузка оператора игнорирует математические свойства фундаментальных операций и выработанную интуицию о природе математических операций.

Виды полиморфизма

- Универсальный
  - параметризация*
  - включение (inclusion, sub-typing)
- Специальный (ad-hoc)
  - перегрузка (overloading)
  - приведение (coercion)

Виды полиморфизма

- Универсальный (единая структура типов)
  - параметризация
  - включение
- Специальный (разная структура типов)
  - перегрузка
  - приведение

Tagged union

```
enum tag { Char, Int, Double };

union types {
    char char_;
    int int_;
    double double_;
};

struct variant {
    tag tag_;
    types type_;
};
```

```
void f(const variant &var) {
    switch(var.tag_) {
    case Char:
        // версия для var.type_.char_
        break;
    case Int:
        // версия для var.type_.int_
        break;
    case Double:
        // версия для var.type_.double_
        break;
    default:
        break;
    }
}
```

Супер-типы и под-типы

```
class abstract_data_t {
public:
    virtual iterator find(int) = 0;
    …
};

class vector: public abstract_data_t {
public:
    iterator find(int) override;
    …
private:
    int *p{nullptr};
};
```

Полиморфизм под-типов

```
vector v = {1, 2, 3, 4};
…
list l = {5, 4, 3, 2, 1};
…
abstract_data_t *a = &v;
abstract_data_t *b = &l;
…
a->find(4);
b->find(4);
```

https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/polymorphism

Приведение типов

Явное:

```
int i = (int)abs(negative);
// или
int i = int(abs(negative));
// или
int i = static_cast<int>(abs(negative));
```

Неявное:

```
int *p = malloc(sizeof(int) * 10);
```

Управление неявным приведением классов в C++

- Создание подходящего **конструктора**, который принимает аргумент заданного типа
- Перегрузка **оператора присваивания**, которая принимает аргумент заданного типа
- Перегрузка **оператора приведения** типа

Управление неявным приведением классов в C++

```
class A { … };

struct B {
    // приведение из типа A (конструктор)
    B(const A &x) {}
    // приведение из типа A (присваивание)
    B& operator=(const A &x) { return *this; }
    // приведение к A
    operator A() { return A(); }
};
```

https://cplusplus.com/doc/tutorial/typecasting/

```
auto x = prod({1, 2, 3}, {4, 5, 6});

auto y = prod({1.0, 2.0}, {4.5, 5.0});

auto z = prod({"a", "b"}, {"c", "d"});
```

Перегрузка функций

```
int prod(int (&a)[10], int (&b)[10]) {
    int k = 0;
    for(size_t i = 0; i < 10; i++) {
        k += a[i] * b[i];
    }
    return k;
}

std::string prod(std::string (&a)[10], std::string(&b)[10]) {
    std::string k = "";
    for(size_t i = 0; i < 10; i++) {
        k += a[i] + b[i];
    }
    return k;
}
```

Параметризация в C++

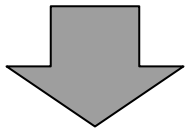
```
template <typename T, size_t n>
auto prod(const T(&a)[n], const T(&b)[n]) {
    T k = 0;
    for(size_t i = 0; i < n; i++) {
        k += a[i] * b[i];
    }
    return k;
}
```

Параметризация в C++

```
template <typename T, size_t n>
auto prod(const T(&a)[n], const T(&b)[n]) {
    T k = T{};
    for(size_t i = 0; i < n; i++) {
        k += a[i] * b[i];
    }
    return k;
}
```

Параметризация в ML

```
fun factorial n =
    if n <= 1 then 1
    else factorial (n-1) * n
```

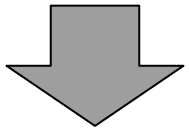


```
val factorial = fn : int -> int
```

Параметризация в ML

```
fun prod (a, b) =
    if a = [] orelse b = []
    then 0
    else hd a * hd b + prod(tl a, tl b)

prod ([1, 2, 3], [4, 5, 6])
```

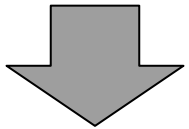


```
val prod = fn: int list * int list -> int
```

Параметризация в ML

```
fun filter pred [] = []
   | filter pred (a::rest) =
   if pred a
   then a::(filter pred rest)
   else (filter pred rest)

filter (fn x => x <> "text") ["text", "notext"]
```

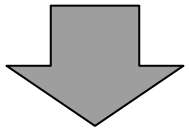


```
val filter = fn : ('a -> bool) -> 'a list -> 'a list
```

Параметризация в ML

```
fun map (f, xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f, xs'))

map (fn x => x + 1) [4, 8, 12, 16]
```



```
val map = fn : ('a -> 'b) * 'a list -> 'b list
```

Функциональное программирование

"No matter what way you work in, programming in a **functional** style provides benefits. You should do it whenever it is convenient, and you should **think hard** about the decision when it is **not** convenient."

J. Carmack

http://number-none.com/blow/john_carmack_on_inlined_code.html

"While it is possible to define object types in any way, there is a set of natural laws that govern the **behavior of most types**. These laws define the meaning of fundamental operations on objects: **construction**, **destruction**, **assignment**, **swap**, **equality** and total **ordering**."

A. Stepanov

Принципы сравнения объектов в C++20

<compare>

1. std::**strong_ordering** (линейно упорядоченное)
2. std::**partial_ordering** (линейно упорядоченное)
3. std::**weak_ordering** (частично упорядоченное)
4. **operator ⇔** (concept **three_way_comparable**)

Частично и линейно упорядоченное множество

ЧУМ (partial order):

Для всех **a, b, c** в **P:**

1. **a <= a**
2. **if a <= b and b <= a then a == b**
3. **if a <= b and b <= c then a <= c**

ЛУМ (total order):

Для всех **a, b, c** в ЧУМ **T:**

1. **a <= a**
2. **if a <= b and b <= a then a == b**
3. **if a <= b and b <= c then a <= c**
4. **a <= b or b <= a**

https://en.wikipedia.org/wiki/Total_order

Примеры

1. **Сравнение рациональных чисел (strong order)**
2. **Сравнение чисел с плавающей запятой (partial order)**
3. **Сравнение точек в декартовой системе координат (weak order)**

https://en.wikipedia.org/wiki/Weak_ordering

Сравнение рациональных чисел

Для **b**, **d** не равных 0:

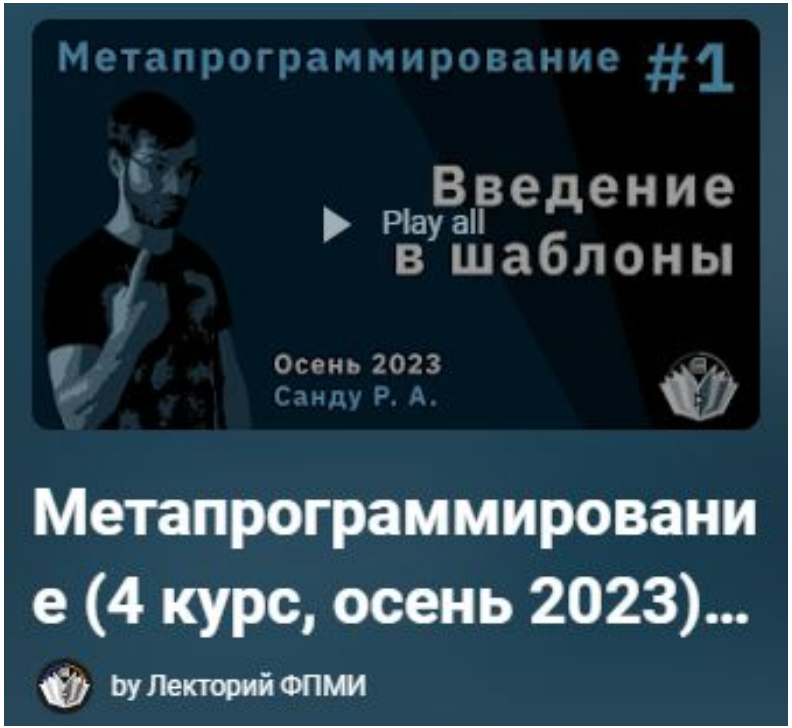
a / b < c / d ⇔ a * d < c * b

Тогда:

```
assert(rational(2, 3) < rational(-3, -4));      // ?
assert(rational(1, 2) <= rational(2, 4));       // ?
```

https://github.com/boostorg/rational/blob/develop/include/boost/rational.hpp#L785

Шаблонное мета-программирование



https://www.youtube.com/playlist?list=PL4_hYwCyhAvYO01i2gR-prnu4Stvxuf7u

Почему в ООП важно то, какая **упорядоченность** у конкретного **типа**?

При чем тут алгебра?

“My math background made me realize that **each object could have several algebras associated with it**, and there could be **families** of these, and that these would be very very useful. The term "*polymorphism*" was imposed much later (I think by Peter Wegner) and **it isn't quite valid**, since it really comes from the nomenclature of functions, and *I wanted* quite a bit *more than functions*. I made up a term "genericity" for dealing with **generic behaviors in a quasi-algebraic form**.”

A. C. Kay

https://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en