Объектно-ориентированное программирование

Object-oriented programming

X. Обобщенное программирование

Generic programming

Fundamentals of Generic Programming

James C. Dehnert and Alexander Stepanov

Silicon Graphics, Inc.
dehnertj@acm.org, stepanov@attlabs.att.com

Keywords: Generic programming, operator semantics, concept, regular type.

Abstract. **Generic programming depends on the decomposition of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces. Among the interfaces of interest, indeed the most pervasively and unconsciously used, are the fundamental operators common to all C++ built-in types, as extended to user-defined types, e.g. copy constructors, assignment, and equality. We investigate the relations which must hold among these operators to preserve consistency with their semantics for the built-in types and with the expectations of programmers. We can produce an axiomatization of these operators which yields the required consistency with built-in types, matches the intuitive expectations of programmers, and also reflects our underlying mathematical expectations.**

- декомпозиция программ на компоненты, разрабатываемые независимо, которые можно комбинировать в произвольном порядке благодаря четко-заданным интерфейсам;
- самые часто используемые интерфейсы – базовые операторы для встроенных типов: **копирование**, **присваивание**, **сравнение**;
- необходима аксиоматизация таких операторов, чтобы сделать работу с пользовательскими типами аналогичной работе со встроенными (интуитивно понятной и отражающей математические свойства.)

Как выглядел мир до обобщенного программирования

Библиотеки являются примером успешно внедренных интерфейсов (Unix, графические библиотеки, математические библиотеки и т.д.)

Библиотеки прошлого используют полностью определенные (конкретные) интерфейсы, поддерживающие заранее заданные (конкретные) типы данных

Плюс библиотек в том, что если ими пользоваться по определенным для них правилам, правильный результат гарантирован

Минус библиотек в том, что пользоваться ими не по заранее определенным правилам (например, для нового типа данных) нельзя

Как это ограничение обойти

Отделение типов данных от структур данных и от алгоритмов позволит комбинировать любые типы с любыми структурами из одной библиотеки и алгоритмами из другой библиотеки

Для этого большое количество компонентов должно иметь общий интерфейс, чтобы быть **взаимозаменяемыми**

Нужно отойти от старой модели и перейти к новой, где четко заданные интерфейсы заменяются на минимальный набор требований к интерфейсам, чтобы разные интерфейсы можно было использовать в одной ситуации

Использование похожих, но разных интерфейсов требует выявления и обобщения шаблонных приемов программирования, а также разработки техник для передачи данных между разными интерфейсами

Concept

“The set of **axioms** satisfied by a *data type* and a set of **operations** on it.”

- целочисленный тип и все операции, удовлетворяющие всем арифметическим свойствам;
- список данных, в котором есть первый элемент, итератор для прохода по списку, способ проверить конец списка;

Любой компонент, который планируется использовать в будущем, должен быть разработан с минимальным набором таких концепций, сами концепции должны подходить наиболее широкому разнообразию структур в программе.

Исторический процесс

Обобщение машинной архитектуры (IBM 360) – вся память устройства представляется как последовательность байтов, обозначающихся однородными адресами (указателями), не зависящими от типа данных.

Обобщение языка конкретной платформы (высокоуровневые языки типа C) – предоставляются составные типы данных на базе обобщенной машинной архитектуры, которые моделируют объекты в памяти, используя указатели для операций, не зависящих от типов данных.

Обобщение встроенных в высокоуровневые языки операций (перегрузка функций и операторов в C++) – можно определять все операции для своих типов с помощью классов и шаблонов.

Из этих трех этапов естественным образом вырастает обобщенное программирование

Суть ОП

“If we hope to reuse code containing references to the standard C++ operators, and apply it to both built-in and user types, we must **extend the semantics** as well as **the syntax** of the standard operators **to user types**. That is, the standard operators must be understood to **implement well-defined concepts** with uniform axioms rather than arbitrary functions.”

Ключ к этому – в использовании семантики указателей для создания концепций, которые повторяют семантику встроенных типов и операторов.

Regular types

Default constructor	`T a;`
Copy constructor	`T a = b;`
Destructor	`~T(a);`
Assignment	`a = b;`
Equality	`a == b`
Inequality	`a != b`
Ordering, e.g.	`a < b`

A **regular type** matches the built-in type semantics, thereby making our **user-defined types behave like built-in types** as well

Фундаментальные операции

Copy, Assignment, and Equality

```
1. T a = b; assert(a == b);

2. T a; a = b; ⇔ T a = b;

3. T a = c; T b = c; a = d; assert(b == c);

4. T a = c; T b = c; zap(a); assert(b == c && a != b);
```

Отношения равенства

Равенство рефлексивно, симметрично и транзитивно, но…

```
bool operator==(const T& x, const T& y) const {
    return true;
}
```

Тогда,

```
x == y ⇔ ∀ predicate P, P(x) == P(y)
```

Отношения равенства

Равенство рефлексивно, симметрично и транзитивно, но…

```
bool operator==(const T& x, const T& y) const {
    return true;
}
```

Тогда,

```
x == y ⇔ ∀ predicate P, P(x) == P(y)
```

Но,

```
∀ predicate P, P(x) == P(y) ⇒ x == y
```

Побитовое сравнение не всегда адекватно

Структуры данных часто состоят из указателей (“**удаленных**” частей) на другие структуры – сравнивать надо не указатели, а структуры, на которые они указывают

Структуры данных часто содержат части, которые не имеют отношения к сущности типа данных, который моделируется – сравнивать такие несущественные части не надо (пример: shared pointer)

Выражения часто состоят из нескольких структур, и нужно иметь возможность определить где заканчивается одна структура и начинается другая – некоторые части структуры определяют отношение своей структуры к какой-то другой структуре, их сравнивать не надо

Равенство двух объектов

Two objects are equal if their corresponding parts are equal (applied recursively), including remote parts (but not comparing their addresses), excluding inessential components, and excluding components which identify related objects.

```
x == y  ⇒ ∀ “reasonable” function foo, foo(x) == foo(y)
```

Например, для двух рациональных чисел:

```
        r1 == r2     ⇒ r1.p == r2.p
    (1, 2) == (2, 4) ⇒    1 == 2
```

Какие функции можно считать “reasonable”?

“For optimization purposes, there are several classes of functions we would like to capture.

First are the **standard operators on built-in types** that do not have side effects, for example **a+b**, **c-d**, or **p%q**.

Second are the **visible member accesses**, e.g. s**.first** or c**->imaginary**.

A third class is the well-known pure functions, e.g. **abs**(x), **sqrt**(y), and **cos**(z).

The ultimate solution, then, must be to **identify the important attributes, and allow programmers to specify them explicitly**.”

https://youtu.be/aXOChLn5ZdQ

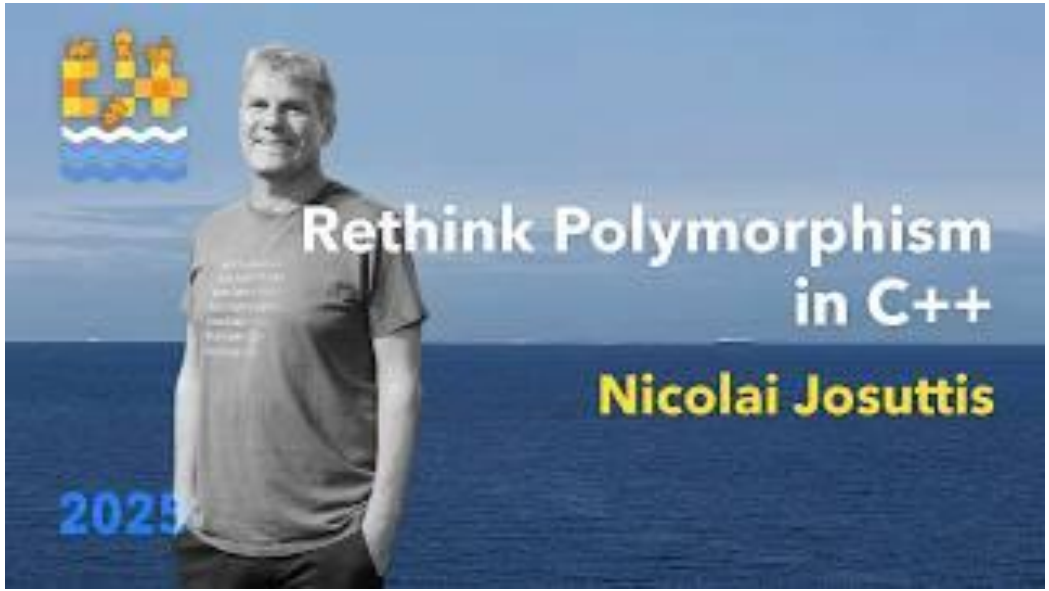
Что стоит иметь ввиду

1. Нестрогое сравнение
2. Ассимптотичность
3. Интуицию, связанную с реализацией встроенных операций, которую программисты вырабатывали годами
4. Общие/ключевые свойства в различных подходах к решению задач одного вида, которые распространены в индустрии
5. Как все это можно выразить в виде аксиом, которые будут соответствовать интуитивным подходам и математическим свойствам

VIII.a. Отношение порядка в C++

Ordering

https://www.youtube.com/watch?v=zI0DOKN6zr0



"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**)

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 точек в декартовой системе координат

In order of increasing strength, i.e., decreasing sets of pairs, three of the possible partial orders on the Cartesian product of two partially ordered sets are:

1. the **lexicographical** order:

   $(a, b) \leq (c, d)$ *if* $a < c$ *or* $(a = c$ *and* $b \leq d)$;

2. the **product** order:

   $(a, b) \leq (c, d)$ *if* $a \leq c$ *and* $b \leq d$;

3. the **reflexive closure** of the direct product of the corresponding strict orders:

   $(a, b) \leq (c, d)$ *if* $(a < c$ *and* $b < d)$ *or* $(a = c$ *and* $b = d)$.

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**

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

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

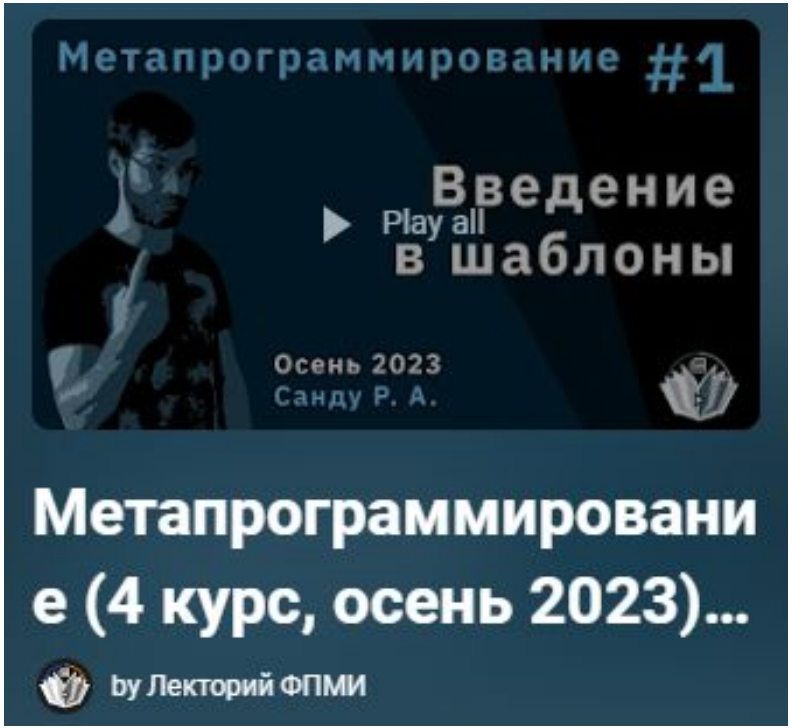
При чем тут алгебра?

“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

Шаблонное мета-программирование



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