Предефиниране на оператори.

Предефиниране на оператори. Приятелски класове и функции. Defaulted comparison operators и <=> (C++20). Ключова дума auto. Пример за реализация на клас Student и School.

д-р Тодор Цонков
todort@uni-sofia.bg

Какво са операторите в C++?

В C++ операторите са символни или ключови конструкции, които извършват операции върху операнди (данни или изрази) и връщат резултат. Те са фундаментален механизъм за манипулиране на данни в езика.

Опростено: операторът е функция със специална синтактична форма, а операндите са аргументите ѝ.

Примери: +, -, *, &&, ==.

Операторите са:
- унарни (с един операнд)
- бинарни (с два операнда).

Всеки оператор се характеризира с:
- Позиция на оператора спрямо операнда (операндите) му;
- Приоритет;
- Асоциативност

В езика C++ не е възможно да бъдат създавани нови оператори,
но са дадени средства за предефиниране на съществуващи

Какво са операторите в C++?

Пример:
Позицията на оператора спрямо операнда (операндите) му го определя като:
- префиксен (операторът е пред единствения си операнд),
- инфиксен (операторът е между двата си операнда)
- постфиксен (операторът е след единствения си операнд).

Закачка: Какво ще изпечати следния код: a+++++b - еквивалентно на:
a++ ++ +b или a++ + ++b?
(Верен отговор 1)

Операторът * е инфиксен (a * b), операторът + е както инфиксен, така и префиксен, а операторът ++ е както постфиксен (a++), така и префиксен (++a).

Приоритетът определя реда на изпълнение на операторите в израз.
Оператор с по-висок приоритет се изпълнява преди оператор с по нисък приоритет.

Какво е предефиниране на оператори?

В C++ много оператори могат да се предефинининират (operator overloading), което позволява дефинирането на собствено поведение за класове.

Пример: за клас Vector можем да дефинираме operator+, така че да събираме два вектора с v1 + v2.

Всеки оператор има приоритет и асоциативност, което определя реда на изпълнение в сложни изрази.

Кодът става по-четим
Обектите се държат като вградени типове

```
Student a, b;
if (a == b) { ... }
```

```
if (a != b) {... }
```

if (a < b) -> какво означава? в случая със Student не става ясно

```
Student operator+(const Student&
a, const Student& b);
```

Какво означава това?

Кога да предефинираме оператори?

1)Когато операторът има естествен смисъл
2)Когато семантиката е ясна
3)Не за „изненадващо“ поведение

Добър пример: ==, <, <<
Лош пример: operator&& с нетипично значение

Не трябва да се променя смисъла на оператора
И трябва да се избира non-member функции, когато е възможно.

```
class Vector2D {
  double x, y;
public:
  Vector2D(double x_, double y_) :
x(x_), y(y_) {}

  // Предефиниране на +
  Vector2D operator+(const
Vector2D& other) const {
      return Vector2D{x + other.x, y +
other.y};
  }
};
```

Предефиниране на оператори

Следните оператори не може да бъдат предефинирани:
. - оператор за избор на член на клас ●
* - оператор за избор на член на клас чрез указател
● :: - оператор за присъединяване - примерно към namespaces
● ?: - троен условен израз
● sizeof

Java не позволява предефиниране на оператори, а в C# е по-ограничено от C++

Предефинираме оператори само когато логиката на класа го изисква, при управление на ресурси, или когато прави кода по-ясен и естествен за използване. Предефиниране без реална нужда е анти-практика и може да въведе неочаквани бъгове.

Ключова дума friend

В C++ friend е ключова дума, която позволява на дадена функция или клас да има достъп до private и protected членове на друг клас, без да бъде негов член.

Това е контролирана форма на нарушаване на инкапсулацията, използвана, когато е логически оправдано.

```
class Box {
  double width;
public:
  Box(double w) : width(w) {}
  // friend функция не е част от
класа!
  friend void printWidth(const Box& b);
};
// Достъп до private член
void printWidth(const Box& b) {
  std::print("Width: {}\n", b.width);
}
```

friend за предефиниране на оператори

```
class Engine {
    int horsepower;
public:
    Engine(int hp) : horsepower(hp) {}
    friend class Car; // Car е приятел
на Engine
};

class Car {
public:
    void showHorsepower(const
Engine& e) {
        std::print("Horsepower: {}\n",
e.horsepower); // достъп до private
    }
};
```

```
class Point {
    int x, y;
public:
    Point(int a, int b) : x(a), y(b) {}
    friend std::ostream&
operator<<(std::ostream& os, const Point& p) {
        os << "(" << p.x << ", " << p.y << ")";
        return os;
    }
};
Point p{3, 4};
std::print("{}\n", p); // ще извика friend
функцията
```

friend за предефиниране на оператори

friend е контролирано изключение от правилата на енкапсулация, използва се за тясно свързани функции или класове, и не трябва да се използва без сериозна причина.

Да се ползва friend рядко и само когато е логично, защото нарушава енкапсулацията на класа.
Да се ползват методи на класа или getter/setter, ако е възможно.

Friend свойството не е транзитивно и реципрочно: Ако клас A е friend на клас B това не означава автоматично, че клас B е friend на клас A.

Съответно, ако A е friend на B, а B е friend на C, това не означава, че A е friend на C.

Defaulted оператор в езика C++(20+)

Трябва да предефинираме всички оператори: ==, !=, <, >, <=, >=
но имаме много boilerplate код.

В C++ operator<=> е т.нар. three-way comparison operator (spaceship operator), въведен в C++20.

Той реализира пълно сравнение между два обекта и връща обект от тип:
std::strong_ordering
std::weak_ordering
std::partial_ordering

```
#include <compare>
#include <string>
class Student {
    std::string name_;
    int age_;
public:
    Student(std::string name, int age)
        : name_(std::move(name)), age_(age) {}

    auto operator<=>(const Student&) const =
default;
};
```

Defaulted оператор в езика продължение

Какво прави компилаторът?

Генерира лексикографско сравнение:
Сравнява name_
Ако са равни → сравнява age_
Връща резултат от сравнение

auto operator<=>(const X&) const = default;

Компилаторът автоматично генерира:
operator==, operator<, operator<=, operator>, operator>=

Компилаторът ще го генерира само ако:
1)всички базови класове са сравними
2)всички членове имат валиден <=>
3)няма ambiguous или deleted comparison
Иначе операторът става имплицитно deleted.

Defaulted <=>:
1) Имплицира value semantics
2) Гарантира консистентност на сравненията
3) Елиминира дублиране на код
4) Намалява грешки при поддържане

Сравнение с C# и Java

C++:
Позволява overloading на почти всички оператори (+, -, *, /, ==, <=>, [], (), -> и др.)
Позволява дефиниране на собствени семантики за вашите типове
Част от compile-time type system
Може да се дефинира като член-функция или friend функция

Java: няма операторно предефиниране
Може да се използват само методи (add(), equals())

Поддържа + за String concatenation като синтактичен sugar

C#: позволява операторно предефиниране, но ограничено до определен набор оператори (+, -, *, /, ==, !=, <, >, <=, >=, true, false, !, ~, ++, --)
В C# операторите трябва да се дефинират в двойка за == и !=, или < и >
Дефинират се като static методи в класа

Ключова дума auto в C++

В C++ думата auto е ключова дума за автоматично извеждане на типове (type deduction).

Той позволява на компилатора да определи типа на променлива на база на нейната инициализация, ако няма типът няма как да бъде изведен.

Тя се използва много често в модерния C++ (C++11 → C++23) и има няколко различни роли, като типът се извежда от компилатора.

```
auto x = 5;       // int
auto y = 3.14;    // double
auto z = "hello";  // const char*
auto add(int a, int b) {
    return a + b;
} //C++ 14 and above
auto multiply(int a, double b) -> double {
    return a * b;
}
```

```
decltype(auto) get(int& x) {
    return x;
} - запазва точния тип, константност
```

Кога се ползва auto в C++

auto извежда типа от инициализатора по същите правила като template type deduction.

```
int a = 5;
int& r = a;
auto x = r;  // int ❌ reference се губи
auto& y = r;    // int&    ✅
```

Да се ползва при:

Дълги и сложни типове:

❌ std::unordered_map<std::string, std::vector<int>>::iterator i

✅ auto it = myMap.begin();

✅ generic код

✅ Iterator и

Когато искаме да обходим колекция:

```
for (auto x : v)     // ❌ копие
for (auto& x : v)    // ✅ reference
for (const auto& x : v) // ✅ най-често правилното
```

Да не се ползва auto, когато:

1)типът не е ясен

2)скрива важна семантика

3)може да доведе до нежелано копиране

auto сравнена с var в C# и Java

1) Премахва се const и & по подразбиране

```
const int a = 10;
auto x = a;     // int (НЕ е const int)
```

Решение:

```
int a = 5;
int& f() { return a; }

auto x = f();         // int   (копие)
decltype(auto) y = f(); // int&  (референция)
```

В езиците C# и Java се използва ключовата дума var.auto x = 5; // C++

var x = 5; // Java // C#

но в C++ може да се използва в много по-широк контекст.

var в Java не е системен механизъм, а удобство и синтактична захар, докато в C++ е част от template type deduction, който е основен механизъм на езика.

В C# е по-мощен, използва се при връщане на функции, LINQ и други

Цялостен пример

```cpp
#pragma once
#include <string>
#include <vector>
#include <iostream>

class Student {
public:
    static constexpr double maxGPA = 4.0;

    // Конструктор с параметри
    Student(const std::string& name, int age)
        : name_(name), age_(age)
    {
        ++studentCount_;
    }

    // Забраняваме copy и move
    Student() = delete;
    ~Student() = default;

    // Getter-и
    const std::string& getName() const { return name_; }
    int getAge() const { return age_; }

    // Setter-и
    void setAge(int age) { age_ = age; }

    static int getStudentCount() { return studentCount_; }

private:
    std::string name_;
    int age_;
    static int studentCount_;
};
```

Цялостен пример

```
// Приятелски клас School
class School {
    std::string school_name;
    std::vector<Student> students;

public:
    explicit School(std::string name) : school_name(std::move(name)) {}

    void add_student(const Student& s) {
        students.push_back(s);
    }

    void print_students() const {
        std::cout << "Students in " << school_name << ":\n";
        for (auto const& s : students) { // auto + range-based for
            std::cout << s << "\n";
        }
    }

    // Приятелска функция, която сравнява две школи по броя на студентите
    friend bool compare_schools(const School& a, const School& b) {
        return a.students.size() < b.students.size();
    }
};
```

Цялостен пример

```cpp
int main() {
    Student s1{"Todor", 21, 3.9};
    Student s2{"Maria", 20, 4.0};
    Student s3{"Ivan", 22, 3.7};

    School school1("Oxford High");
    school1.add_student(s1);
    school1.add_student(s2);

    School school2("Cambridge Academy");
    school2.add_student(s3);

    school1.print_students();
    school2.print_students();

    // Използваме приятелската функция
    if (compare_schools(school1, school2)) {
        std::cout << school1.get_name() << " has fewer students.\n";
    } else {
        std::cout << school2.get_name() << " has fewer students.\n";
    }

    // Пример за сравнение на студенти чрез <=>
    if (s1 < s2) {
        std::cout << s1.name << " has lower GPA than " << s2.name << "\n";
    }
}
```

Въпроси?

Благодаря за вниманието!

Допълнителни материали: learncpp.com глава 15, 10