Обектно-ориентирано програмиране (записки)

- **Марина Господинова**
- **Илиан Запрянов**

Тема 07. Предефиниране на оператори. Приятелски класове и функции.

def| оператор - функция със специален синтаксис

Операнд - нещото, върху което операторите извършват своето действие - константи, променливи, изрази, функции

Видове оператори в C++ има **3 вида оператори**:
Унарни оператори - действат върху един операнд

Примери:
- унарен минус (**-**), който променя знака на числото
- унарен плюс (**+**), който запазва знака
- логическо отрицание (**!**), което обръща булевата стойност (например, **`!true`** е **`false`**).
- (**++**) и (**--**), които увеличават или намаляват стойността с единица

Бинарни оператори - действат върху два операнда

Примери
- оператори като събиране (**+**), изваждане (**-**), умножение (*****), и деление (**/**).
- логически оператори като И (**&&**), ИЛИ (**||**).
- оператори за сравнение като равно (**==**), по-голямо (**>**), по-малко (**<**).

Тернарен оператор - има един единствен тернарен оператор (действа между три операнда)
- условният оператор, изразен като **условие** ? **израз1** : **израз2**, където ако условието е истина (**true**), резултатът е **израз1**; ако е лъжа (**false**), резултатът е **израз2**

```
int a = 0;
int b = 0;

//унарни оператори
a++;
-b;
//бинарни оператори
a + b;
if (a && b) { /*...*/ };
//тернарният оператор
a > b ? std::cout << "plovdiv" : std::cout << "plovdiv";
```

Характеристики на оператори

Приоритет на операторите - реда, в който операциите се изпълняват (операторите с по-висок приоритет се изпълняват преди тези с по-нисък)

Например:

```
int a = 3;
int b = 4;
int c = 5;

//[*] > [+] (приоритет)
std::cout << a + b * c << std::endl; //23: 3 + 4 * 5 = 2 + 20 = 23
//                 ^
//                 |
//операторът между [b] и [c] се извършва преди този между [a] + [b] заради приоритета
```

Асоциативност - определя реда, в който операторите с еднакъв приоритет се изпълняват (всички оператори с еднакъв приоритет са или **ляво-асоциативни**, или **дясно-асоциативни**, т.е. няма оператори с еднакъв приоритет, при които единият да е ляво-асоциативен (от ляво надясно), а другият дясно-асоциативен (от дясно наляво))

- **ляво-асоциативни:** операторите се изпълняват от ляво надясно (оператори като събиране **(‘+’)**, умножение **(‘*’)**, логическо И **(‘&&”)**

- **дясно-асоциативни:** операторите се изпълняват от дясно наляво (оператори като събиране **(‘+=’)**, умножение **(‘*=’)**, логическо И **(‘&=”)**

```
int a = 3;
int b = 4;
int c = 5;

//[+] е ляво-асоциативен
//------------------------------>
std::cout << a + b + c << std::endl; //12: 3 + 4 + 5 = 7 + 5 = 12

//[=] е дясно-асоциативен
//<------------------------------
a = b = c; //a = b = c -> b = c ([b] става 5)
           //          -> a = b ([b] е станало 5, [a] става 5)

std::cout << a; // 5
```

Позиция - разположението на оператора спрямо операндите

- **префиксни** - оператори, които стоят **ПРЕД** операнда
- **постфиксни** - оператори, които стоят **СЛЕД** операнда
- **инфиксни** - оператори, които са разположени **МЕЖДУ ДВА ОПЕРАНДА**

```
int a = 3;
int b = 4;
int c = 5;

//префиксен оператор
++a;

//инфиксен оператор
a + b;

//постфиксен оператор
a++;

return 0;
```

Когато говорим за **обекти** (непримитивни типове), то нямаме дефинирани оператори като събиране **(‘+’)**, умножение **(‘*’)**, а трябва да ги дефинираме сами, което се извършва чрез създаването на нови функции (**външни и вътрешни за класа**), които симулират действието на даден оператор.

Ще използваме следната структура

```
struct A
{
	int a1 = 0;
	int a2 = 0;

public:
	A(int a1, int a2) : a1(a1), a2(a2)
	{
		;
	}

	void printAll() const
	{
		std::cout << a1 << " " << a2 << std::endl;
	}
};
```

Първи начин на предефиниране на оператор за произволен оператор (който ще означим с **$**) - **външна функция**

```
friend A& operator$(const A& other); //ще го разгледаме по-късно (декларация в тялото на [A])

A operator$(const A& a1, const A& a2) //подаваме два обекта
{
	this->a1; //[X] функцията е външна, тоест нямаме достъп до член-данните на класа,
		  //тъй като щом е външна не сочи към свой собствен обект,
		  //а просто работи с  такива обекти, което означава, че нито един от обектите
		  //формално не извиква оператора (тоест не се извиква от конкретен обект, тъй
		  //като функцията не се свързва (асоциира) с такъв)

		  //когато кажем, че функцията е friend (приятелска) това не променя това
		  //(ще разгледаме friend по-късно)

	//логика, която да симулира действието на произволния оператор [$]
	//
	//
}
```

Първи начин на предефиниране на оператор за произволен оператор (който ще означим с **$**) - **вътрешна функция**

```
A& A::operator$(const A& other) //подаваме ЕДИН обект
{                               //(имаме предвид, че скришно се подава *this най-вляво),
                                //тъй като е ВЪТРЕШНА, означава че е член-функция =>
                                //има свой собствен обект, което означава,
                                //че ЛЕВИЯТ обект е този, който извиква оператора

	this->a1 + other.a1; //имаме достъп до член-данните на this (текущия обект)

	//логика, която да симулира действието на произволния оператор [$]
	//
	//
}
```

Уговорка:

- операторите, които **ПРОМЕНЯТ** левия обект ще изнасяме във **ВЪТРЕШНИ** функции
- операторите, които **НЕ ПРОМЕНЯТ** левия обект ще изнасяме във **ВЪНШНИ** функции

При предефиниране на оператори трябва да се спазват **общоприети стандарти**

```
#include <iostream>

struct A
{
	int a1 = 0;
	int a2 = 0;

public:
	A(int a1, int a2) : a1(a1), a2(a2)
	{
		;
	}

	void printAll() const
	{
		std::cout << a1 << " " << a2 << std::endl;
	}

	A& operator+=(const A& other);

};
```

```
A& A::operator+=(const A& other) //операторът [+=] ще симулира действието на прибавянето
                                 //на стойностите на член-данните
{                                //на десния обект към левия, тоест ЩЕ СЕ промени левия
                                 //=> по уговорка ВЪТРЕШНА функция
	//добавяме стойностите
	//на втория обект към първия

	this->a1 += other.a1;        //връщането по референция (не void функция)
	this->a2 += other.a2;        // ни позволява така нареченото,
                                 //chain-ване на оператори, т.е. ако имаме три обекта от
                                 //тип [A] -> [first], [second], [third], то
                                 //става валиден синтаксиса
                                 //[first] += [second] += [third]

                                //напомняме, че ако връщаме по копие, chain-ването също
                                //е възможно, но връщаме по референция с цел да
                                //направим кода си по-ефективен
	return *this;
}
```

```
A operator+(const A& first, const A& second) //операторът [+] ще симулира действието събиране
                                             //между два обекта, тоест НЯМА ДА СЕ промени левия
{                                            //=> по уговорка ВЪНШНА функция

      /тъй като вече сме предефинирали
      //оператора [+=] можем да го преизползваме

      //създаваме третия обект, който ще връщаме
      //директо чрез копиращия конструктор
      //(така данните му ще вземат стойностите на един от тях)
      A res(first);

                                    //напомняме, че при ВЪНШНО-предефинираните оператори, ще
                                    //връщаме ПО КОПИЕ, тъй като във функцията създаваме
                                    //трети обект, който да върнем, т.е. локален за функцията обект
                                    // => ще се изтрие след скоупа на функцията =>
                                    //ако върнем по референция, ще имаме референция към вече
                                    //несъществуващ обект, което ще доведе до [UB]

      //преизползваме [+=]
      //(променяме левия и му добавяме стойностите на десния)

      res += second;
      return res;
}
```

```
bool operator==(const A& first, const A& second) //левия не се променя => ВЪНШНА ФУНКЦИЯ
{                                                //при предефиниране на operator== е прието да
                                                 //връщаме тип [bool], тъй като операторът не е
      //примерен стандарт за еквивалентност      //предназначен за chain-ване =>
      return first.a1 == second.a1               // трябва просто да върнем [true], ако два обекта са
          && first.a2 == second.a2;              //еквивалентни по даден стандарт и [false], ако не са
}
```

Особени случаи

1. Оператор<< и оператор>>

```
std::ostream& operator<<(std::ostream& os); //нека първоначално operator<< ни е обикновена
                                            //член-функция (напомняме, че уговорката тук НЕ важи,
                                            //тъй като е за оператори между два обекта от ЕДИН ТИП,
                                            //тук операторът е между ПОТОК и ОБЕКТ

std::ostream& A::operator<<(std::ostream& ofs) //връщаме по референция по аналогични причини на [+=]
{                                              //(по-ефективно chain-ване)

                                               //помним, че при член-функции като първи аргумент
                                               //винаги се подава скришно пойнтъра [this], който
                                               //сочи към текущия обект, тоест можем да си представим
                                               //параметрите като (<обекта>, <потока>)

                                               //
                                               //какво променя това?


      return ofs << this->a1 << " " << this->a2; //връщаме потока след като запазим в него [a1], [a2]
}




int main()
{
      A a(2, 3);

      //тъй като скришно се подава this,
      //това означава, че ПЪРВО се подава this, ВТОРО - потока,
      //което води до обръщането на стандартния синтаксис, с който сме свикнали

      //[X] std::cout << a; (стана невалиден синтаксис)
      a << std::cout;
      //(this) << (<поток>) (реда, в който са ни параметрите на дефинираната функция operator<<)
      //        ^
      //        |
      //     оператор

      return 0;
}
```

```
    //за да се придържаме към стандартния синтаксис
    //ще изнесем operator<< във външна функция, за да избегнем
    //скришното подаване на [this], което се извършва във всички
    //вътрешни функции, то тогава ще имаме параметри (<поток>, <обект>),
    //за разлика от вътрешната функция, която показахме

    //за да достъпим член-данните на обекта (които са private/protected)
    //във външната функция, ще кажем, че тя е friend (приятелска за обекта)

    friend std::ostream& operator<<(std::ostream& os, const A& obj);
```

```
std::ostream& operator<<(std::ostream& ofs, const A& obj) //връщаме по референция по аналогични
{                                                         //причини на [+=] (по-ефективно chain-ване)

                                                  //не подаваме тайно [this], т.е. се придържаме към
                                                  //синтаксиса, на който сме свикнали


    return ofs << obj.a1 << " " << obj.a2; //връщаме потока след като запазим в него [a1], [a2]
}
```

```
int main()
{
	A a(2, 3);

	//тъй като направихме функцията външна,
	//това означава, че ПЪРВО се подава потока, ВТОРО - this,

	std::cout << a;
	//[X] a << std::cout; (стана невалиден синтаксис)
	//(<поток>) <<  (this) (реда, в който са ни параметрите на дефинираната функция operator<<)
	//          ^
	//          |
	//      оператор

	return 0;
}
```

```
//функцията за operator>> е аналогична
friend std::istream& operator>>(std::istream& is, A& obj); //за да се придържаме към
                                                           //стандартния синтаксис
```

```
std::istream& operator>>(std::istream& is,  A& obj) //[NOTE] : забелязваме, че потоците, които
{                                                   //връщаме и подаваме са ostream и istream
                                                    //(това е така, защото НЕ работим с файлове,
                                                    //а с всякакви потоци (поддържащи вход/изход)
	return is >> obj.a1 >> obj.a2;
}
```

```
int main()
{
	A a(2, 3);

	std::cin >> a;

	return 0;
}
```

Предефиниране на оператори за имплементация и деплементация

a++ -> връща старата стойност **(a - 1)**, т.е. увеличава **a** с единица, но функцията на оператора връща стойност **a -1**
++a -> замества се с новото **a** (новата стойност), т.е. увеличава **a** с единица и оператора връща новата стойност на **a**

```
A& operator++(); //променяме обекта => вътрешни
A operator++(int);
```

```
//забелязваме, че prefix-ния ++ (++a) се връща по референция, докато
//postfix-ния ++ (a++) се връща по копие

//това идва от дефиницията на двата оператора в C++,
//която води до това, че prefix-ния може да се chain-ва,
//докато postfix-ния - не, т.е. [++++a] е валидно, но [a++++] - не

//идеята на postfix-ния е да върне старата стойност на [a],
//а на prefix-ния - новата =>
//postfix прави копие на [a] преди да го увеличи и връща КОПИЕТО,
//prefix увеличава [a] и го връща

A& A::operator++()
{
	this->a1++;
	this->a2++;

	return *this; //връщаме текущия обект с НОВИТЕ му стойности  (така работи prefix-ния ++)
}

A A::operator++(int) //int се нарича dummy parameter (параметър, който не се използва)
{                    //той на практика не изпълнява никаква функционална роля освен
                     //да помогне за различаването между префиксната и постфиксната форма на
                     //оператора, т.е. служи за отбелязка, която ни помага
                     //да различим [a++] и [++a]
	this->a1++;
	this->a2++;

	return A(this->a1 - 1, this->a2 - 1); //връщаме СТАРИТЕ стойности
                                          //(така работи postfix-ния ++)
}




int main()
{
	A a(2, 3);

	std::cout << a++.a1 << " " << ++a.a2 << std::endl; //2 5 (първо връща старата стойност на
                                                       //[a1] и увеличава [a] с 1 (postfix)
                                                       //=> имаме предвид, че [a1], [a2] се
                                                       //увеличават с [1] и [a2] става [4],
                                                       //след което увеличаваме [a] още веднъж
                                                       //(prefix) => [a2] се увеличава с [1],
                                                       //става на [5]и го отпечатваме

	return 0;
}
```

Забележки:

- някои оператори **ТРЯБВА** да са член-функции - **=**, **[]**, **()**, **->**
  - с **()** можем да викаме обектите като **функции** (въпрос за теория), т.е. **obj(); е викане на функция**
- **ограничения**
  - някой оператори **не могат** да се предефинират: **? :** (тернарния оператор), **::** (оператор за резолюция), **.** (оператор за достъп)
  - нови оператори **НЕ** могат да се създават (**$** в горния пример е невалиден оператор)
  - **НЕ** можем да предефинираме характеристиките на операторите - асоциативност/приоритет/позиция спрямо аргументите
  - оператора **->** трябва да връща **обект (указател/референция/копие)**
  - при предефиниране на **&&**,**||** губим **early exit** (предварително изчисляване), т.е.
    - при **&&** не спира при първо грешно (не прави **false** оценка веднага)
    - при **||** не спира при първо вярно (не прави **true** оценка веднага)

Демото, разписано за предефиниране на операторите:

https://github.com/Zapryanovx/FMI_2024_Object_Oriented_Programming/tree/main/00_demos/Operators%20Redefinition

Приятелски класове/функции

def| класове/функции, които имат достъп до private имплементациите ни

```
#include <iostream>

class B
{
	A obj;
public:
	B()
	{
		obj.x = 7; //тъй като класът [B] е приятелски на [A]
	};		   //можем директно да достъпим private член-данната [x] на [obj]

	friend void g();

	//...

};
class A
{
	int x;
public:
	A() = default;

	friend void f();
	friend class B;

	//friend void g(); //за да имаме достъп до private член-данните на [A] в [g]
			   //трябва да кажем, че тя е приятелска за този клас

	//...
};
```

```
void f()
{
	A obj;
	obj.x++; //аналогично на [B], във функцията [f] също можем да достъпваме private
		 //член-данните на [obj]
}

//приятелят на моя приятел НЕ Е мой приятел:

//класът [A] има приятелски клас [B], който има приятелска функция [g],
//това обаче НЕ ни дава достъп до private член-данните на [A] във функцията [g]
void g()
{
	A obj;
	//obj.x++;
}

int main()
{


	return 0;
}
```