Предефиниране на оператори

Характеристики на операторите в C++:

- брой операнди:
  - унарен (с един операнд)
  - бинарен (с два операнда)
  - тернарен (с три операнда) (един единствен в езика)
- позиция на оператора спрямо аргументите:
  - префиксен (разположен е пред единствения си аргумент)
  - инфиксен (операторът е между аргументите си)
  - постфиксен (операторът е след единствения си аргумент)
- приоритет - определя реда на изпълнение на операторите (тези с по-вискок приеоритет се изпълняват преди тези с по-нисък)
- асоциативност - определя реда на изпълнение на операторите с еднакъв приоритет:
  - лявоасоциативни - изпълняват се от ляво надясно
  - дясноасоциативни - изпълняват се от дясно наляво

Таблица на операторите в C++, http://www.cplusplus.com/doc/tutorial/operators/

From greatest to smalles priority

Level	Precedence group	Operator	Description	Grouping	Overloading
1	Scope	`::`	scope qualifier	Left-to-right	NO
2	Postfix (unary)	`++ --`	postfix increment / decrement	Left-to-right	YES
		`()`	functional forms	Left-to-right	YES
		`[]`	subscript	Left-to-right	YES
		`. ->`	member access	Left-to-right	`.` - NO `->` - YES
3	Prefix (unary)	`++ --`	prefix increment / decrement	Right-to-left	YES
		`~ !`	bitwise NOT / logical NOT	Right-to-left	YES

Level	Precedence group	Operator	Description	Grouping	Overloading
		`+ -`	unary prefix	Right-to-left	YES
		`& *`	reference / dereference	Right-to-left	YES
		`new delete`	allocation / deallocation	Right-to-left	YES
		`sizeof`	parameter pack	Right-to-left	NO
		`(type)`	C-style type-casting	Right-to-left	NA
4	Pointer-to-member	`.* ->*`	access pointer	Left-to-right	`.*` - NO `->*` - YES
5	Arithmetic: scaling	`* / %`	multiply, divide, modulo	Left-to-right	YES
6	Arithmetic: addition	`+ -`	addition, subtraction	Left-to-right	YES
7	Bitwise shift	`<< >>`	shift left, shift right	Left-to-right	YES
8	Relational	`< > <= >=`	comparison operators	Left-to-right	YES
9	Equality	`== !=`	equality / inequality	Left-to-right	YES
10	And	`&`	bitwise AND	Left-to-right	YES
11	Exclusive or	`^`	bitwise XOR	Left-to-right	YES
12	Inclusive or	`|`	bitwise OR	Left-to-right	YES
13	Conjunction	`&&`	logical AND	Left-to-right	YES
14	Disjunction	`||`	logical OR	Left-to-right	YES
15	Assignment-level expressions	`=` `*= /= %=` `+= -=` `>>=` `<<= &= ^=` `|=`	assignment / compound assignment	Right-to-left	YES

Level	Precedence group	Operator	Description	Grouping	Overloading
		`?:`	conditional operator	Right-to-left	NO
16	Sequencing	`,`	comma separator	Left-to-right	YES

В C++:

- **НЕ могат** да се създават нови оператори
- **Могат** да се предефинират съществуващите
- Предефинираният оператор запазва всички характеристики на оригиналния, т.е. не могат да му се променят *приоритетът*, *асоциативността* и *броят и позицията* на аргументите

Операторите:

- `=`
- `+`, `-`, `*`, `/`, `%`,
- `+=`, `-=`, `*=`, `/=`, `%=`,
- `++`, `--`,
- `^`, `&`, `|`, `~`,
- `<<`, `>>`
- `^=`, `&=`, `|=`,
- `<`, `>`, `==`, `!=`, `<=`, `>=`,
- `>>=`, `<<=`,
- `&&`, `||`, `!`,
- `->*`, `->`, `[]`, `()`, `,`,
- `new`, `delete` , `new []`, `delete []`

могат да бъдат предефинирани, стига поне един операнд на оператора да е обект на някакъв клас (повечето интуитивно подсказват какви са им характеристиките, за непознатите в интернет пише всичко).

Изключения са:

- Оператори, на които вторият операнд е **име**, а не **стойност**:
  - `::` - scope operator (оператор за обхват)
  - `.` - оператор за достъп до компонента на клас
  - `.*` - оператор за достъп до компонента на клас чрез указател
- Оператори с по-специална специфика:
  - `?:` - оператор за троен условен израз (тернарният оператор)
  - `sizeof` - оператор за размера на обект или тип

и не могат да бъдат предефинирани.

Предефиниране на оператори в C++:

Осъществява с чрез дефиниране на специален вид функции, които се наричат операторни.

Те имат синтаксис подобен на синтаксиса на обикновените функции, но името им се състои от ключовата дума `operator`, следвана от означението на предефинирания оператор. (пример е предефинирането на оператор =, за който сме говорили)

```
Student& operator=(const Student& other_object);
```

Когато предефинирането на оператор изисква достъп до компонентите на клас, обявени като `private` или `protected`, операторната дефиниция трябва да е или метод на класа или приятелска функция на класа.

Нека да дадем пример с класа ComplexNumber (по-долу във файла е целия код):

*** Относно примерите едно уточнение. Имената на аргументите, които се падат и операнди, когато предефинираме оператори ги означавам *lhs* и *rhs*, което означава съответно Left Hand Side и Right Hand Side и съответнно представят лявостоящия и дясностоящия операнд кода.

- като метод на класа:

```
class ComplexNumber{
public:
    ComplexNumber operator+(const ComplexNumber& rhs) const;
};

ComplexNumber ComplexNumber::operator+(const ComplexNumber& rhs) const{
    ComplexNumber result;
    result.real = this->real + rhs.real;
    result.imaginary = this->imaginary + rhs.imaginary;
    return result;
}
```

- като приятелска функция:

```
class ComplexNumber{
friend ComplexNumber operator+(const ComlexNumber& lhs, const ComplexNumber&
rhs);
};

ComlexNumber operator+(const ComplexNumber& lhs, const ComplexNumber& rhs){
    ComplexNumber result;
    result.real = lhs.real + rhs.real;
    result.imaginary = lhs.imaginary + rhs.imaginary;
    return result;
}
```

- така го бяхме направили преди да стане въпрос за предефиниране на оператори

```
ComplexNumber sum(const ComplexNumber& rhs) const;

ComplexNumber ComplexNumber::sum(const ComplexNumber& rhs) const{
    ComplexNumber result;
    result.real = this->real + rhs.real;
    result.imaginary = this->imaginary + rhs.imaginary;
    return result;
}
```

Ограничения:

- Когато първият (или единственият) операнд на оператор, който искаме да предефинираме, е обект на класа или референция към обект на класа, операторът може да се предефинира и като метод на класа и като приятелска функция на класа.
  - Предефинирането на операторите `()`, `[]`, `->` и `=` трябва да е като методи на класа. Причината е, че така се гарантира първият им аргумент да е лявостоящата стойност.
  - При предефиниране като методи на класа, първият (или единственият) операнд не се задава като параметър. Ролята му се изпълнява от сояения от указателя `this` обект.
- Когато първият (или единственият) операнд на оператор, който искаме да предефинираме, е обект на друг клас или псевдоним на обект на друг клас, или е от стандартен в езика тим (*int*, *char*, стрийм и др.), операторът трябва да се реализира като приятелска функция на класа. Като ще предефинираме операторите `>>` и `<<` и използваме поток за първи операнд

Да започнем да разглеждаме спецификите при предефиниране на различните типове оператори:

- Унарните оператори могат да се предефинират и чрез метод на класа и чрез приятелска функция.

  Пример ще дадем с класа ComplexNumber и ще предефинираме *унарния префиксен* оператор `-` (унарния, не бинарния), който ще връща комплексно число, което е с противоположни знаци за реалната и имагинерната част.

  Отново можем да го направим и по двата начина, въпрос на стил и избор от ваша страна.

```
class ComplexNumber{
    ComplexNumber operator-() const;//ще работи върху this, т.е. текущият обект и затова няма аргумент (казахме, че компилатора си го добавя в последствие this като аргумент)
};

ComplexNumber::ComplexNumber operator-() const{
    double real = -this->real;
    double imaginary = -this->imaginary;
    return ComplexNumber(real, imaginary);
}
```

```
class ComplexNumber{
    friend ComplexNumber operator-(const ComplexNumber& rhs);
};

ComplexNumber operator-(const ComplexNumber& rhs){
    double real = -rhs.real;
    double imaginary = -rhs.imaginary;
    return ComplexNumber(real, imaginary);
}
```

- Унарните оператори `++` и `--` са и префиксни, и постфиксни. (Добре е да се знае как работят префиксния и постфиксния вариант в C++, ако има въпроси в интернет бързо ще намерите отговора, иначе ми пишете).

В такъв случай, дефинирането им се случва по различен начин. Иначе, отново става и двата варианта на предефиниране на операторната ункция - чрез метод или приятелска функция. Ако искаме да предефинираме `++` или `--` като:

- префиксен оператор(`++a` , `--a`) - предефинираме по подобие на другите префиксни оператори, както досега (без аргументи, ако операторната функция е метод на класа, или с един аргумент, ако е приятелска функция)

  ```
  ComplexNumber& operator++();

  ComplexNumber& ComplexNumber::operator++(){
      cout << "Prefics ++\n";
      this->real++;
      return *this;
  }
  ```

  ***Имплементацията я правя да инкрементиа само реалната част

- постфиксен оператор(`a++` , `a--`) - предефинираме, като в операторните функции се използва допълнителен аргумент от тип *int* (той няма да върши нищо полезно, просто означение за компилатора, наричат го още dummy аргумент). Когато компилаторът види `a++` или `a--`, тогава ще извика предефинирания оператор ++ или ––, който има аргумент *int* и ще му подаде на аргумента стойност 0. Иначе тази стойност е без значение, аргументът е dummy, само да обозначи за компилатора коя предефинирана функция да извика.

  ```
  ComplexNumber operator++(int);

  ComplexNumber ComplexNumber::operator++(int){
      cout << "Postfix ++\n";
      ComplexNumber temp = *this;
      this->real++;
      return temp;
  }
  ```

- Бинарни оператори (могат да бъдат предефинирани като методи на класа или като приятелски функции, даже може и като външни операторни функции, но това не препоръчителен начин):

  Вече дадохме по-нагоре примера с оператор +, който е аритметичен, да пробваме и с бинарни релационни оператори един пример

  - като метод на класа

    ```
    bool operator==(const ComplexNumber& rhs) const;

    bool ComplexNumber::operator==(const ComplexNumber& rhs) const{
        return (this->real == rhs.real && this->imaginary == rhs.imaginary);
    }
    ```

  - като приятелска функция

```
friend bool operator==(const CompelxNumber& lhs, const ComplexNumber&
rhs);

bool operator==(const CompelxNumber& lhs, const ComplexNumber& rhs){
    return (lhs.real == rhs.real && lhs.imaginary == rhs.imaginary);
}
```

Тук ви качвам целия код от примения клас ComplexNumber.

```
#include <iostream>
using namespace std;

class ComplexNumber {
private:
    double real;
    double imaginary;
public:
    ComplexNumber();
    ComplexNumber(double real, double imaginary);
    void setReal(double real);
    void setImaginary(double imaginary);
    double getReal() const;
    double getImaginary() const;
    ComplexNumber sum(const ComplexNumber& rhs) const;
    ComplexNumber multiply(ComplexNumber rhs) const;

    ComplexNumber& operator++();
    ComplexNumber operator++(int);

    bool operator==(const ComplexNumber& rhs) const;

    void print() const;

friend ComplexNumber operator+(const ComplexNumber& lhs, const ComplexNumber&
rhs);
friend ComplexNumber operator-(const ComplexNumber& rhs);
};
```

```
#include "ComplexNumber.hpp"

ComplexNumber::ComplexNumber(){
    this->real = 0;
    this->imaginary = 0;
}

ComplexNumber::ComplexNumber(double real, double imaginary){
    this->real = real;
    this->imaginary = imaginary;
}

void ComplexNumber::setReal(double real){
    this->real = real;
```

```
}
void ComplexNumber::setImaginary(double imaginary){
    this->imaginary = imaginary;
}
double ComplexNumber::getReal() const{
    return real;
}
double ComplexNumber::getImaginary() const{
    return imaginary;
}


ComplexNumber ComplexNumber::sum(const ComplexNumber& rhs) const{
    ComplexNumber result;
    result.real = this->real + rhs.real;
    result.imaginary = this->imaginary + rhs.imaginary;
    return result;
}



ComplexNumber ComplexNumber::multiply(ComplexNumber rhs) const{
    ComplexNumber result;
    result.real = this->real*rhs.real - this->imaginary*rhs.imaginary;
    result.imaginary = this->real*rhs.imaginary + this->imaginary*rhs.real;
    return result;
}

ComplexNumber& ComplexNumber::operator++(){
    cout << "Prefics ++\n";
    this->real++;
    return *this;
}


ComplexNumber ComplexNumber::operator++(int){
    cout << "Postfix ++\n";
    ComplexNumber temp = *this;
    this->real++;
    return temp;
}

bool ComplexNumber::operator==(const ComplexNumber& rhs) const{
    return (this->real == rhs.real && this->imaginary == rhs.imaginary);
}

void ComplexNumber::print() const{
    if(imaginary >= 0){
        cout << real << "+" << imaginary << "i" << "\n";
    }
    else{
        cout << real << imaginary << "i" << "\n";
    }
}

ComplexNumber operator+(const ComplexNumber& lhs, const ComplexNumber& rhs){
    ComplexNumber result;
    result.real = lhs.real + rhs.real;
```

```
    result.imaginary = lhs.imaginary + rhs.imaginary;
    return result;
}

ComplexNumber operator-(const ComplexNumber& rhs){
    double real = -rhs.real;
    double imaginary = -rhs.imaginary;
    return ComplexNumber(real, imaginary);
}
```

```
#include <iostream>
#include "ComplexNumber.hpp"

int main() {
    ComplexNumber a;
    a.setReal(4);
    a.setImaginary(2);

    ComplexNumber b;
    b.setReal(3);
    b.setImaginary(5);

    a.print();
    b.print();

    ComplexNumber res1 = a.sum(b);
    res1.print();

    ComplexNumber res2 = a.multiply(b);
    res2.print();

    (a + b).print();

    (-res2).print();

    (++res2).print();//check prefics

    (res2++).print();//check postfix

    res2.print();//check postfix (should be incremented after the last res++)

    ComplexNumber p;

    cout << (p == a) << "\n";//not equal

    p = a;//assign to p the value of a

    cout << (p == a) << "\n";//equal

    p.operator++().print(); //just to show they work like normal functions

    return 0;
}
```