

Тема 7 | Преобразуване на оператори. Приятелски класове и ф-ции

Оператори

- ф-ции са специални симболи
- operator X ($X \in$ оператор)

Пример

```
class A {int n;}
```

```
A obj1, obj2;
```

```
obj1 + obj2;
```

Характеризират се с:

- асоциативност - при изрази с оператори от еднакъв тип га се дефинира "последователност" на операциите

а) право $((a \$ b) \$ c) \$ d \rightarrow (+, -, *, /)$

б) ляво $(a \$ (b \$ (c \$ d))) \rightarrow (=)$

- приоритет - при израз с повече от един оператор определя последователността на изпълнение

Пример: $* > / > +$

- позиция (на он спрямо аргументите)

а) префиксни $+a, -a, !a, \sim a$

б) инфиксни $a+b, a-c$

в) постфиксни (съфиксни) $a-; a++$

△ Не можем да предизвикате несъществуващи оператори (да създадеш нов)

△ Не можем да променяме приоритета, асоциативността и подчинената на оператор

△ Не можем да предизвикате следните от: `,(.)`, `(::)`, `(?:)`, `(sizeof)`, `(alignof)`, `(noexcept)`

Тип на операторите

- унарни (1 аргумент) `++a`, `--a`, `+a`, `-a`, `*a`
- бинарни (2 аргумента) `a+b`, `a-b`, `a/b`, `a^b`.
- Тернарен - той е само `?:`
- при бинарните от 1, лявата операция се приема като първи параметър, а дясната като втори

△ Изключение прави `op()` (function call), който може да приема произволен брой

△ При предизвикателство на `&` и `||` губим short-circuit eval

△ Три наличнища на поредка на тройба от, какът и другите (`>`, `<`, `=`, `==`, `!=`) те могат да използват от <

△ Три проверки за реверсивно члене от `=` и `!=`

△ Три предизвикателства тройба да запази return type

- чин зе преобразуване
- бъдуща функция

- идентификация на private член-члените
- идентификация на константите и член-члените
- променливите не са наричани.

Пример:

Complex operator+ (const Complex & l, const Complex & r)
Complex n(l);
 $n += r$.
return n; $\rightarrow \underline{\underline{NRVO}}$

- }
- член об-лии
- идентификация на private член-члените
- локални аргументи е this

Пример: Complex operator+= (const Complex & r)
- real+=r._real;
- im+=r._im;
return *this;

⚠ $=, -=, /=, *=$ са гет. като член-об-лии
 $+, -, /, */$ са гет. като бъдущи об-лии

⚠ Има функции, които могат ~~само~~ да са същите
базични: (=), (≤), (()), (→), (*) и т.н.

Преобразуването на он за икономичните ++, --

++ a - връща променливата current value

a++ - връща стапата current value

Пример:

```
class X {  
    int n;
```

```
X & operator++() {  
    + +n;  
    return *this;  
}
```

```
X operator++(int dummy) {  
    X temp = *this;  
    + +n;  
    return temp;
```

Преобразуване на I/O он.

- когато имаме I/O неизв. от класа трябва
да преобразувам он за него.

Пример:

```
A obj;  
cin >> obj;  
cout << obj;
```

- трябва да епъят istream</stream за да
може да има chaining

Пример: ostream & operator<<(ostream& os, const A& obj) {
 >> return os;

istream & operator >>(istream& is, A& obj)

{
return is;

}

⚠ Тези функции трябва да имат 1ъв аргумент
потока, да са външни, залагато са общи.

Пример: istream & operator >>(istream &){...}

A obj;

obj >> cin; // не отваря ли структурата,
но, тези трябва и да могат да променят полетата
на ѝ

⇒ функциите трябва да е външни и да
променят полетата ⇒ friend функции

Приятелски класове и функции

- класове/функции върнати за клас, който имат
достъп до private член-даници на класа

- friend

- не са привилегирани, не се наследяват

- норма значение ѝ са private/public/protected
такъгда

Пример: struct A {

 friend istream & operator >>(istream, A &){...}

 class SharedPtr{

 struct Counter{...}

 friend class WeakPtr;

(5)

NVector.h

```
pragm a once  
+include <iostream>
```

```
class NVector
```

```
{ private:
```

```
    int* data=nullptr;
```

```
    size_t size=0;
```

```
    void copyFrom(const NVector& other);
```

```
    void free();
```

```
public:
```

```
    NVector(size_t size);
```

```
    NVector(const NVector& other);
```

```
    NVector& operator=(const NVector& other);
```

```
    ~NVector();
```

```
    NVector& operator+= (const NVector& other);
```

```
    NVector& operator-= (const NVector& other);
```

```
    NVector& operator*= (size_t scalar);
```

```
    int& operator[](size_t);
```

```
    int operator[](size_t) const;
```

```
    size_t operator~() const;
```

friend std::ostream & operator<<(std::ostream & o, const NVector & v);

friend std::ifstream & operator>>(std::ifstream & i, NVector & v);

};

NVector operator+(const NVector & lhs, const NVector & rhs);

NVector operator-(const NVector & lhs, const NVector & rhs);

NVector operator*(const NVector & lhs, size_t scalar);

NVector operator*(size_t scalar, const NVector & rhs);

bool operator||(const NVector & lhs, const NVector & rhs);

size_t operator%(const NVector & lhs, const NVector & rhs);

bool operator/=(const NVector & lhs, const NVector & rhs);

1/1/2014
finclude "NVector.h"

NVector.cpp

```
NVector::NVector(size_t n): size(n)
{
    data = new int[n];
    for (int i=0; i<n; i++)
        data[i] = 0;
}
```

```
NVector::NVector(const NVector & other)
```

```
{ copyFrom(other);
}
```

```
NVector & NVector::operator=(const NVector & other)
```

```
{ if (this != & other)
```

```
{ free();
}
```

```
copyFrom(other);
}
```

```
return *this;
}
```

```
NVector::~NVector()
```

```
{ free();
}
```

```
void NVector::copyFrom(const NVector & other)
```

```
{ size = other.size;
```

```
data = new int[size];
```

```
for (int i=0; i<size; i++)

```

```
    data[i] = other.data[i];
}
```

```
void NVector::free()
{
    delete [] data; data=nullptr;
    size=0;
}
```

```
NVector& NVector::operator+=(const NVector& other)
{
    if(size != other.size)
        throw "The vectors should have the same size!";
    for(int i=0; i<size; i++)
        data[i] += other.data[i];
    return *this;
}
```

```
NVector& NVector::operator-=(const NVector& other)
{
    if(size != other.size)
        throw "The vectors should have the same size!";
    for(int i=0; i<size; i++)
        data[i] -= other.data[i];
    return *this;
}
```

```
NVector operator+(const NVector& lhs, const NVector& rhs)
{
    NVector lhsCopy(lhs);
    lhsCopy+=rhs;
    return lhsCopy;
}
```

NVector operator-(const NVector & lhs,
const NVector & rhs)

| NVector.cpp
| N=2

{ NVector lhsCopy(lhs);
lhsCopy -= rhs;
return lhsCopy;

}

int& NVector::operator[](size_t index)

{ if(index >= size)
throw "Invalid index";
return data[index];

}

int NVector::operator[](size_t index) const

{ if(index >= size)
throw "Invalid index";
return data[index];

size_t NVector::operator[]() const

{ return size;

std::ostream& operator<<(std::ostream& os, const NVector& v)
{ os << "[";
for(int i=0; i < v.size; i++)
os << v[i] << " ";
os << "]";
return os;

```
std::istream & operator>>/std::istream & is, Nvector  
for (int i=0; i<rsize; i++)  
    is >> v[i];  
return is;
```

```
bool operator//(const Nvector & lhs, const Nvector & rhs)  
{  
    if (~lhs != ~rhs)  
        throw "The vectors should have the same size";  
    double ratio=0.0;  
    bool ratioSet=false;  
    for (size_t i=0; i<~lhs; ++i)  
    {  
        if (lhs[i]==0 && rhs[i]==0)  
            continue;  
        if (lhs[i]==0 || rhs[i]==0)  
            return false;  
        double currentRatio=(double)lhs[i]/(double)rhs[i];  
        if (!ratioSet)  
        {  
            ratio=currentRatio;  
            ratioSet=true;  
        }  
        else if (std::abs(ratio-currentRatio)>0.000001)  
            return false;  
    }  
    return true;
```

NVector & NVector::operator*=(size_t scalar) {

NVector.cpp
No 3

```
{ for(int i=0; i<size; i++)  
    data[i] *= scalar;  
    return *this;  
}
```

NVector operator*(const NVector & lhs, size_t scalar)

```
{ NVector copy(lhs);  
copy *= scalar;  
return copy;
```

NVector operator*(size_t scalar, const NVector & rhs)

```
{ return rhs * scalar;
```

NVector operator%(const NVector & lhs, const NVector & rhs)

```
{ if(~lhs != ~rhs)  
    throw "The vectors should have the same size!";
```

```
size_t res = 0;  
for(int i=0; i<~lhs; i++)  
    res += lhs[i] * rhs[i];
```

```
return res;
```

```
}
```

bool operator !=(const NVector & lhs, const NVector & rhs) {
 {
 return lhs % rhs != 0;
 }
}

```
#pragma once
```

```
#include <iostream>
```

```
class Complex
```

```
{ private:
```

```
    double real = 0.0;
```

```
    double im = 0.0;
```

```
public:
```

```
    Complex() = default;
```

```
    Complex(double real, double im);
```

```
    double getReal() const;
```

```
    double getIm() const;
```

```
    void setReal(double real);
```

```
    void setIm(double im);
```

```
    Complex getConjugated() const;
```

```
    Complex& op+=(const Complex&);
```

```
    Complex& op-=(const Complex&);
```

```
    Complex& op*=(const Complex&);
```

```
    Complex& op/=(const Complex&);
```

```
    friend std::ostream& operator<<(std::ostream&, const Complex&);
```

```
    friend std::istream& operator>>(std::istream&, Complex&);
```

```
}
```

Complex operator + (const Complex&, const Complex&);
Complex operator - (const Complex&, const Complex&);
Complex operator * (const Complex&, const Complex&);
Complex operator / (const Complex&, const Complex&);

bool operator == (const Complex& lhs, const Complex& rhs);

bool operator != (const Complex& lhs, const Complex& rhs);

```


#include "Complex.h"
Complex::Complex(double real, double im): real(real), im(im) {}

Complex& Complex::operator+=(const Complex& other)
{
    real += other.real;
    im += other.im;
    return *this;
}

Complex& Complex::operator-=(const Complex& other)
{
    real -= other.real;
    im -= other.im;
    return *this;
}

Complex& Complex::operator*=(const Complex& other)
{
    double oldReal = real;
    real = real * other.real - im * other.im;
    im = oldReal * other.im + im * other.real;
    return *this;
}

Complex& Complex::operator/=(const Complex& other)
{
    Complex Conjugated = other.getConjugated();
    Complex otherCopy(other);
    *this *= Conjugated;
    other *= Conjugated;
    if(otherCopy.real != 0)
        if(real != otherCopy.real || im != otherCopy.real)
            return *this;
}


```

Complex.cpp

Complex operator+(const Complex& lhs, const Complex& rhs)

{

 Complex result(lhs);
 result += rhs;
 return result;

}

Complex op-(const Complex& lhs, const Complex& rhs)

{

 Complex result(lhs);
 result -= rhs;
 return result;

}

Complex op*(const Complex& lhs, const Complex& rhs)

{

 Complex result(lhs);
 lhs *= rhs;
 return result;

}

Complex op/(const Complex& lhs, const Complex& rhs)

{

 Complex result(lhs);
 lhs /= rhs;
 return result;

}

Complex Complex::getConjugated() const

{

 Complex result(*this);
 result.im *= -1;
 return result;

}

File Complex::getReal() const

Complex.cpp
No2

return real;

double Complex::getIm() const

return im;

void Complex::setReal(double newReal)

{ real = newReal;

}

void Complex::setIm(double newIm)

{ im = newIm;

std::ostream& operator<</std::ostream& os, const Complex&r)

{ return os << r.real << ' ' << r.im << 'i';

std::istream& operator>>(std::istream& is, Complex&r)

{ return is >> r.real >> r.im;

bool operator==(const Complex&lhs, const Complex&rhs)

{ return lhs.getReal() == rhs.getReal() && lhs.getIm() == rhs.getIm();

bool operator!=(const Complex&lhs, const Complex&rhs)

{ return !(lhs == rhs);