


Laborator #5

Supraîncărcarea operatorilor

Standard Template Library

Supraîncărcarea operatorilor

Supraîncărcarea operatorilor

 Supraîncărcarea operatorilor = mecanism pentru personalizarea funcționalității operatorilor pentru tipurile de date definite de utilizator (clase/structuri/templates).

Operatorii supraîncărcați sunt, în esență, funcții cu nume speciale, supraîncărcate. Ei pot aparține clasei pentru care sunt definiți sau pot fi non-membri.

Supraîncărcarea operatorilor

Ce operatori pot fi supraîncărcați?

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

Ce operatori nu pot fi supraîncărcați?

.	.*	::	?:
---	----	----	----

Supraîncărcarea operatorilor

Implementări canonice: operatorii supraîncărcați ar trebui să aibă un comportament cât mai asemănător posibil cu implementarea implicită, dată de limbaj. Operatori supraîncărcați în mod uzual:

1. operatorul de asignare (cu cazurile particulare: copiere, mutare): `=`;
2. operatorii de extragere/insertie din/în fluxuri: `<<`, `>>`
3. operatorii de incrementare/decrementare: `++`, `--`
4. operatorii aritmetici binari: `+`, `-`, `*`, `/`
5. operatorii relaționali: `<`, `>`, `<=`, `>=`, `==`, `!=`

Supraîncărcarea operatorilor

```
#include <iostream>
class Complex{
private:
    float re;
    float im;
public:
    Complex(float re=0, float im=0):re(re), im(im){}

    Complex operator+(const Complex &c){
        return Complex(this->re+c.re, this->im+c.im);
    }

    Complex operator/(float scalar){
        if (scalar != 0){
            return Complex(this->re/scalar, this->im/scalar);
        }
        else {
            std::cout << "Impartire la 0";
            exit(1);
        }
    }

    void afisare(){
        std::cout << this->re << " + i*" << this->im;
    }
};
```

```
int main(){
    float re;
    float im;

    std::cin >> re >> im;
    Complex c1(re, im);

    std::cin >> re >> im;
    Complex c2(re, im);

    ((c1 + c2) / 2).afisare();

    return 0;
}
```

Supraîncărcarea operatorilor << și >>

```
#include <iostream>

class Complex{
private:
    float re, im;
public:
    Complex(float re=0, float im=0):re(re), im(im){}
    friend std::ostream& operator<<(std::ostream& os, const Complex& c);
    friend std::istream& operator>>(std::istream& is, Complex& c);
};

std::ostream& operator<<(std::ostream& os, const Complex& c){
    os << c.re << " + i*" << c.im;
    return os;
}

std::istream& operator>>(std::istream& is, Complex& c){
    is >> c.re >> c.im;
    return is;
}
```

```
int main(){
    Complex c;

    std::cin >> c;
    std::cout << c;

    return 0;
}
```

Supraîncărcarea operatorilor relaționali

```
#include <iostream>

class Complex{
private:
    float re, im;
public:
    Complex(float re=0, float im=0):re(re), im(im){}
    friend bool operator<(const Complex& lhs, const Complex& rhs);
    friend bool operator>(const Complex& lhs, const Complex& rhs);
    friend bool operator<=(const Complex& lhs, const Complex& rhs);
    friend bool operator>=(const Complex& lhs, const Complex& rhs);
    ... // supraîncărcarea operatorilor >>, ++
};
```

```
bool operator<(const Complex& lhs, const Complex& rhs){
    return ((lhs.re<rhs.re) && (lhs.im<rhs.im));
}
bool operator>(const Complex& lhs, const Complex& rhs){
    return rhs<lhs;
}
bool operator<=(const Complex& lhs, const Complex& rhs){
    return !(rhs<lhs);
}
bool operator>=(const Complex& lhs, const Complex& rhs){
    return !(lhs<rhs);
}
```


```
int main(){
    Complex c;
    std::cin >> c;
    Complex a = c;
    c++;

    std::cout << (a>c) << std::endl;
    std::cout << (a<c) << std::endl;
    std::cout << (a<=c) << std::endl;
    std::cout << (a>=c) << std::endl;

    return 0;
}
```


Standard Template Library

Templates

 Templates sunt un mecanism al limbajului C++ care permite claselor și funcțiilor să opereze cu tipuri de date generice. Templates sunt declarate o singură dată și instanțele template se generează în momentul compilării programului.


Sintaxă:

parantezele unghiulare <> fac parte din sintaxa

template <lista_parametri> declaratie



Standard Template Library

 Standard Template Library (STL) = o bibliotecă de template-uri (clase și funcții), utilizate frecvent în rezolvarea problemelor de programare.

Componentă:

1. Containere (containers)
2. Iteratori (iterators)
3. Algoritmi (algorithms)

4. Obiecte funcții (functors)
5. Alocatori (allocators)

1. Containere (Containers)

Containererele conțin mai multe subtipuri:

1. **Containerere de secvențe (sequence containers):** `array`, `vector`, `deque`, `forward_list`, `list`
2. **Containerere asociative (associative containers):** `set`, `map`, `multiset`, `multimap`
3. **Containerere asociative neordonate (unordered associative containers):** `unordered_set`, `unordered_map`, `unordered_multiset`, `unordered_multimap`
4. **Adaptori de containere (container adaptors):** `stack`, `queue`, `priority_queue`

2. Iteratori (Iterators)

 Iteratori = obiecte care pointează (indică) către elementele din interiorul unui container.

Iteratorii trebuie să implementeze cel puțin 2 funcționalități:

1. iterație – trebuie să poată trece de la un element la altul, prin incrementarea iteratorului (`operator++`);
2. dereferențiere – trebuie să poată citi din memorie elementul către care indică iteratorul (`operator*`).

Cea mai evidentă formă a unui iterator: pointer.

Fiecare container are un iterator specific.

3. Algoritmi (algorithms)

 def Algoritmi = bibliotecă de funcții pentru numeroase mecanisme

“Câțiva” algoritmi importanți:

<code>for_each</code>	<code>count</code>	<code>find</code>	<code>search</code>	<code>copy</code>
<code>move</code>	<code>fill</code>	<code>transform</code>	<code>remove</code>	<code>replace</code>
<code>swap</code>	<code>reverse</code>	<code>sample</code>	<code>unique</code>	<code>is_sorted</code>
<code>sort</code>	<code>partial_sort</code>	<code>binary_search</code>	<code>merge</code>	<code>includes</code>
<code>set_difference</code>	<code>set_intersection</code>	<code>set_union</code>	<code>make_heap</code>	<code>push_heap</code>
<code>pop_heap</code>	<code>sort_heap</code>	<code>max</code>	<code>max_elem</code>	<code>min</code>
<code>min_elem</code>	<code>minmax</code>	<code>clamp</code>	<code>equal</code>	<code>accumulate</code>

Mai multe detalii aici: <https://en.cppreference.com/w/cpp/algorithm>

std::sort

```
template<class RandomIt, class Compare> void sort(RandomIt first,  
RandomIt last, Compare comp);
```

Sortează elementele din intervalul [first, last) în ordine crescătoare (nu strict crescătoare).

Elementele sunt comparate cu ajutorul funcției binare de comparație comp.

În lipsa unei funcții de comparare, se folosește implicit operator<.

std::sort – exemplu

```
#include <algorithm>
#include <vector>
#include <iostream>

void print(std::vector<int> v) {
    for (auto a : v) {
        std::cout << a << " ";
    }
    std::cout << std::endl;
}

bool mai_mic(int a, int b) {
    return a < b;
}
```

```
int main()
{
    std::vector<int> vec;
    vec.push_back(15);
    vec.push_back(20);
    vec.push_back(7);
    vec.push_back(13);

    print(vec);

    std::sort(vec.begin(), vec.end());
    std::cout << "Sortat implicit cu operator<\n";
    print(vec);

    std::sort(vec.begin(), vec.end(), mai_mic);
    std::cout << "Sortat cu functia mai_mic\n";
    print(vec);

    return 0;
}
```


Sfârșit laborator #5