

LABORATORUL 4

1 Proprietăți, metode și obiecte const

Putem declara o variabilă ca fiind constantă (a cărei valoare nu se modifică) adăugând cuvântul cheie **const** înainte de a specifica tipul de date. Când declarăm o variabilă **const** compilatorul va încerca să nu aloce spațiu în memorie pentru acea variabilă.

```
1 int main () {
2     const int i = 3;
3     // i = 4;      eroare de compilare
4     int j = i; // constanta se comporta ca orice variabila de tip intreg
5     return 0;
6 }
```

Cuvântul cheie **const** poate fi amestecat cu pointeri și referințe. Putem declara pointer către constante, pointeri constanți dar și referințe către constante:

```
1 int main () {
2     int i = 6;
3     int b = 3;
4     const int *p = &i; // pointer catre o constanta de tip intreg
5     // *p = 2;          eroare
6     p = &b;             // functioneaza
7     int* const cp = &i; // pointer constant
8     *cp = 6;           // functioneaza
9     // cp = &b;         eroare
10    const int &r1 = i; // referinta catre o constanta
11    return 0;
12 }
```

Într-o clasă putem avea proprietăți și metode **const**. Proprietățile constante pot fi inițializate cu parametri dați către constructor doar în lista de inițializare. Metodele **const** sunt metode care nu pot modifica starea obiectului, i.e. nu pot altera nici una dintre proprietățile obiectului. O metodă poate fi declarată **const** prin adăugarea cuvântului cheie **const** după lista de parametri. Exemplu:

```
1 class A {
2     const int i;
3     float f;
4 public:
5     A (int a = 0, float b = 3.0) : i(a) {
6         f = b;
7         // i = (int) f - a; eroare de compilare
8     }
9
10    void foo () const { // metoda const
11        int j = i + 22;
12        // f = 33;      eroare de compilare
13    }
14
15    void bar () {
16        f = 25;          // metoda nu e constanta, putem modifica
17    }
18 };
```

Odată definită o clasă, putem declara obiecte constante de tipul clasei respective. Cu un obiect constant nu putem apela decât metode **const**.

```
1 int main () {
2     const A a;
3     // a.bar();   eroare de compilare
4     a.foo();     // functioneaza
5 }
```

2 Parametrii default

În C++ putem atribui valori default pentru parametrii unei funcții. La momentul apelării dacă o valoare nu este furnizată pentru un parametru, valoare default este folosită.

Exemplu

```
1 #include <iostream>
2
3 int sum (int a, int b = 0, c = 0) {
4     return a + b;
5 }
6
7 int main () {
8     // se apeleaza sum cu a = 4, b = 87 si c = 34
9     std::cout << sum(4, 87, 34);
10    // se apeleaza sum cu a = 10, b = 43 si c = 0 - valoare default
11    std::cout << sum(10, 43);
12    // se apeleaza sum cu a = 5, b = 0 si c = 0 - b si c valori default
13    std::cout << sum(5);
14
15    return 0;
16 }
```

Observații:

- parametrii default se poziționează mereu la capătul listei de parametri
- De asemenea, nu putem alterna între parametri default și parametri non-default (această semnătură este greșită:
`int foo (int a = 1, int b);`)
- Nu putem alege care dintre parametrii funcției noastre să primească valori default și care nu (în cazul funcției `sum` dacă apelul se face cu doi parametri, atunci parametrul `c` este cel care primește valoarea default).

3 Polimorfism

În C++ avem polimorfism pe funcții și metode. Prin asta înțelegem că putem să re folosim nume de funcții și metode pentru a declara noi funcții și metode, atât timp cât nu avem 2 definiții care au liste identice de parametri (i.e. pentru a supraîncărca/redeclara o funcție trebuie să furnizăm o nouă listă de parametri). Exemplu:

```
1 // Polimorfism funcții
2 #include <iostream>
3 using namespace std;
4
5 int multiply(int a, int b) {
6     return a * b;
7 }
8
9 string multiply (int a, string b) {
10     string result = "";
11     for (int i = 0; i < a; i++) {
12         result += b;
13     }
14     return result;
15 }
16
17 // o schimbare doar tipului de parametri
18 // va rezulta int-o eroare de compilare
19 // float multiply (int a, int b) {
20 //     return float(a)*b;
21 // }
22
23 int main () {
24     int x = multiply(3, 5);
25     string s = multiply(4, "A");
26
27     cout << x << " " << s << endl; // 15 AAAA
28     return 0;
29 }
```

```
1 // Polimorfism metode
2 #include <iostream>
3 using namespace std;
4
```

```

5 class C {
6     int a;
7     public:
8     C(int);
9     int multiply (int);
10    string multiply (string);
11    // float multiply (int); - eroare de compilare
12 };
13
14 C::C(int x) : a(x) { }
15
16 int C::multiply (int b) {
17     return a * b;
18 }
19
20 string C::multiply (string s) {
21     string result = "";
22     for (int i = 0; i < a; i++) {
23         result += s;
24     }
25     return result;
26 }
27
28 int main () {
29     C c (3);
30     cout << c.multiply(5) << " " << c.multiply("A");    // 15 AAA
31     return 0;
32 }

```

4 Supraîncărcarea operatorilor

Datorită polimorfismului putem supraîncărca operatori pentru clase definite de utilizator. Supraîncărcarea nu presupune că putem schimba numărul de parametri (aritatea) și nici precedența operatorului. Toți operatorii cunoscuți pot fi supraîncărcați cu excepția:

- `.` ;
- `::` ;
- `?:` ;
- `sizeof` ;

Când supraîncărcăm majoritatea operatorilor avem două opțiuni: 1. supraîncărcare ca metodă (putem vedea operatorul ca pe o metodă unde operandul din stânga este cel cu care se apelează metoda, iar operandul din dreapta este parametrul metodei) sau 2. supraîncărcare ca funcție prieten. Dacă, la supraîncărcare, primul operand nu are tipul clasei pentru care supraîncărcăm, atunci suntem obligați să supraîncărcăm ca funcție prieten. Dacă nu supraîncărcăm nici un operator, primim de la compilator operatorul `=`.

```

1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4 class Point {
5     int x, y;
6     public:
7     Point (const int& a = 0, const int& b = 0) : x(a), y(b) { }
8     // supraîncărcarea operatorului - pentru a
9     // determina distanța dintre două puncte
10    double operator-(const Point&) const;
11    // supraîncărcarea operatorului * pentru produsul
12    // scalar a două puncte/doi vectori
13    int operator*(const Point&) const;
14    // supraîncărcarea operatorului + pentru
15    // translatarea unui punct
16    friend Point operator* (const int&, const Point&);
17    // supraîncărcarea operatorului << pentru afisarea unui punct
18    friend ostream& operator<<(ostream&, const Point&);
19    // supraîncărcarea operatorului >> pentru citirea unui punct
20    friend istream& operator>>(istream&, Point&);
21 };
22
23 double Point::operator- (const Point& p) const {

```

```

24     double dx = x - p.x, dy = y - p.y;
25     double px = dx*dx, py = dy*dy;
26     return sqrt(px+py);
27 }
28
29 int Point::operator* (const Point& p ) const {
30     return x * p.x + y * p.y;
31 };
32
33 Point operator* (const int& x, const Point& p) {
34     Point o;
35     o.x = x * p.x;
36     o.y = x * p.y;
37     return o;
38 }
39
40 ostream& operator<<(ostream& out, const Point& p) {
41     out << "(" << p.x << ", " << p.y << ")";
42     return out;
43 }
44
45
46 istream& operator>>(istream& in, Point& p) {
47     in >> p.x >> p.y;
48     return in;
49 }
50
51 int main () {
52     Point m(1, 6), n(12, 5);
53     cout << "Distanța dintre " << m << " și " << n << " este " << m - n << endl;
54     // Distanța dintre (1,6) și (12,5) este 11.0454
55     cout << "Translatam " << m << " cu 3 " << 3 * m << endl;
56     // Translatam (1,6) cu 3 (3,18)
57     cout << "Produsul scalar " << m * n << endl;
58     // Produsul scalar 42
59     return 0;
60 }

```

Exerciții

Pentru acest laborator folosiți cât mai multe noțiuni din laboratoarele precedente. În acest laborator nu este permisă folosirea claselor prieten.

1. Implementați clasa **Fraction** (fracție) cu două câmpuri **n** (numărător) și **m** () de tip **int** . Implementați constructorul cu parametri
2. Supraîncărcați operatorii **+** , **-** , ***** și **/** pentru adunarea, scăderea, înmulțirea și împărțirea fracțiilor.
3. Supraîncărcați operatorii de citire (**>>**) și afișare (**<<**) pentru clasa **Fraction** .
4. Implementați amplificarea unei fracții folosind operatorul ***** .
5. Implementați clasa **Vector** , vector de fracții. Implementați constructor și destructor.
7. Implementați metode pentru inserare, ștergere și verificare dacă un element este în vector (**push** , **pop** / **remove** , **contains**).
6. Implementați operatorul de atribuire pentru clasa **Vector** .
7. Implementați operațiile de adunare și scădere pentru clasa **Vector** .
8. Implementați citirea și afisarea pentru clasa **Vector** .

9. Implementați înmulțirea unui `Vector` cu un scalar.
10. Implementați operatorul `[]` pentru accesarea elementului de pe poziția `i`.
11. Implementați metode `countGreater` și `countLower` care întorc numărul de elemente strict mai mari, respectiv strict mai mici, decât o valoare dată.