Supraîncărcarea operatorilor

Introducere

- Un **tip de date** (predefinit) defineşte o mulțime de valori şi o mulțime de operații ce se pot efectua cu acestea. De exemplu: *adunare* (‘+’), *scădere* (‘−’) etc.
- Am putea defini anumite operații cu ajutorul operatorilor, în cazul tipurilor de date definite prin intermediul claselor?

Exemplu: Operații cu numere complexe

```
class Complex {
  private:
    float re;
    float im;
  public:
    Complex (float re = 0, float im = 0);
    void afisare();
    Complex adunare(Complex z);
    Complex conjugatul();
    friend Complex diferenta(Complex z1,
                             Complex z2);
};

Complex::Complex (float re, float im) {
  this->re = re;
  this->im = im;
}

void Complex::afisare() {
  printf("%-g%+g*i\n", re, im);
}
```

```
Complex Complex::adunare(Complex z) {
   Complex rez;
   rez.re = this->re + z.re;
   rez.im = this->im + z.im;
   return rez;
}

Complex Complex::conjugatul() {
  return Complex(re, -im);
}

Complex diferenta(Complex z1, Complex z2) {
  return Complex(z1.re-z2.re, z1.im-z2.im);
}

int main() {
  Complex z1(2,1), z2(3,4), z;
  z = z1.adunare(z2);
  z = diferenta(z1,z2);
  z = z1.conjugatul();
  return 0;
}
```

Alternativa…

- O exprimare mai elegantă a operațiilor:

```
z = z1.adunare(z2);
z = diferenta(z1, z2);
z = z1.conjugatul();
```

- se poate realiza în limbajul C++ prin **supraîncărcarea operatorilor** ‘+’, ‘–’, ‘~’ astfel:

```
z = z1 + z2;
z = z1 - z2;
z = ~z1;
```

Conversii implicite

- *Conversie implicită nedegradantă* (**promovare**): mecanism prin care un tip de date T1 este transformat în alt tip de date T2 care are proprietatea că orice valoare reprezentabilă în T1 poate fi reprezentată în T2.
  - `char`, `short`, `unsigned char` şi `unsigned short` sunt convertite la `int`
  - `bool` este convertit la `int`
  - `float` este convertit la `double`
  - `double` este convertit la `long double`
  - orice enumerare este convertită la `int`.
- *Promovările* sunt folosite şi în cadrul operațiilor aritmetice între tipuri diferite (de ex. `char + int` va rezulta într-un `int`). În acest caz regula este să se utilizeze tipul cel mai mare dintre cele implicate în expresie (pe cât posibil).

Supraîncărcarea funcțiilor

- *Supraîncărcarea* (*overloading*): declararea mai multor funcții cu acelaşi nume, dar care diferă prin tipul şi / sau numărul parametrilor.
- *Supraîncărcarea* operatorilor permite, în general, atribuirea mai multor semnificații aceluiași simbol.
- În practică, un operator sau o funcție este definită de mai multe ori, diferența între definiri fiind dată de *tipul parametrilor şi numărul lor* - niciodată de tipul returnat.
- *Supraîncărcarea* este o *primă formă de polimorfism* (efectuează operații diferite în funcție de context).
- De exemplu, operatorul '+' efectuează adunarea atât a unor numere întregi cât și a unor numere reale.

Supraîncărcarea funcțiilor

- Se încearcă selectarea funcției după numărul de parametri.
- Dacă acest lucru nu se poate efectua în mod neambiguu, atunci selecția se va face după tipul lor, în următoarea ordine:
  - se încearcă identificarea funcției fără efectuarea vreunei conversii de tip;
  - se încearcă *conversii implicite nedegradante* sau *promovare* (*fără pierdere de informații*);
  - se încearcă *conversii implicite degradante* (*cu pierdere de informații*);
  - se apelează eventuale *conversii (explicite) definite de programator* prin supraîncărcarea *operatorului de cast*.

Supraîncărcarea funcțiilor

```
#include <iostream>
using namespace std;
int suma(int a, int b) {
  return a + b;
}
double suma(double a, double b) {
  return a + b;
}

int main() {
  int u = 5;
  float x = 3.12f;
  char ch = 'a';
  long long v = 75000011;

  cout << suma(u, u) << '\n';     // apel suma(int, int)
  cout << suma(x, x) << '\n';     // apel suma(double, double)
  cout << suma(ch, ch) << '\n';   // apel suma(int, int)
  cout << suma(v, v) << '\n';     // apel ambiguu suma(int, int)
                                  // sau suma(double, double)

  return 0;
}
```

Supraîncărcarea funcțiilor

- Căutarea se oprește când o singură funcție coincide apelului.
- Dacă mai multe funcții corespund apelului de funcție atunci vom primi *eroare de ambiguitate*.
- Dacă niciuna nu corespunde apelului de funcție atunci vom avea *eroare de linkeditare.*
- *Supraîncărcarea funcțiilor membre* are sens doar atunci când acestea *fac parte din aceeași clasă* (distincția între clase se face prin operatorul de rezoluție ‘ : : ’).

Supraîncărcarea funcțiilor

- *Funcțiile cu număr variabil de parametri* pot fi considerate de două tipuri:
  - *funcții de fallback* - în linii mari funcții de tipul `name(...)` - nu au decât *parametri variadici*.
    - aceste funcții sunt ultimele apelate, şi sunt utilizate doar dacă toate celelalte modalități de apel au eşuat (*promovare* / *conversie*).
    - dacă apare un caz ambiguu în urma conversiei – nu se ajunge la o astfel de funcție, iar compilatorul va arunca o eroare. Aceste funcții nu sunt permise în C!
  - funcții care au minim un parametru normal şi la final un parametru variadic.
    - Aceste funcții se comportă exact ca şi funcțiile normale (fără parametri variadici).

Supraîncărcarea operatorilor

- Este procesul de atribuire a două sau mai multor operații aceluiaşi operator.
- Se poate realiza prin:
  - *funcție membru, şi*
  - *funcție friend globală*.

Supraîncărcarea operatorilor

- O funcție membră a unei clase poate avea următorul antet:

```
tip operator x();
tip operator x(tip param);
```

- O funcție prietenă, externă clasei **`A`**, poate avea următorul antet:

```
tip operator x(A ob);
tip operator x(A ob, tip param);
tip operator x(tip param, A ob);
```

- unde `tip` este un tip de date (predefinit sau definit de către utilizator), `A` este numele unei clase, `param` şi `ob` sunt variabile de tipul `tip` şi `A`.

Supraîncărcarea operatorilor. Restricții

- Prin supraîncărcarea operatorilor nu se poate modifica:
  - **aritatea operatorilor** (numărul operanzilor asupra cărora se aplică). Astfel, operatorii unari nu pot fi supraîncărcați ca operatori binari şi invers.
  - **asociativitatea operatorilor,**
  - **prioritatea operatorilor.**
- Se pot supraîncărca numai operatorii deja existenți.
- Nu pot fi supraîncărcați operatorii '`.*`', '`::`', '`.`', '`?:`', '`sizeof`'.

Supraîncărcarea operatorilor

```
Complex a, b;
```

`a + b` echivalent cu `a.operator+(b);`

`++a` echivalent cu `a.operator++();`

`a++` echivalent cu `a.operator++(int);`

`2 + a` echivalent cu `operator+(2,a);`

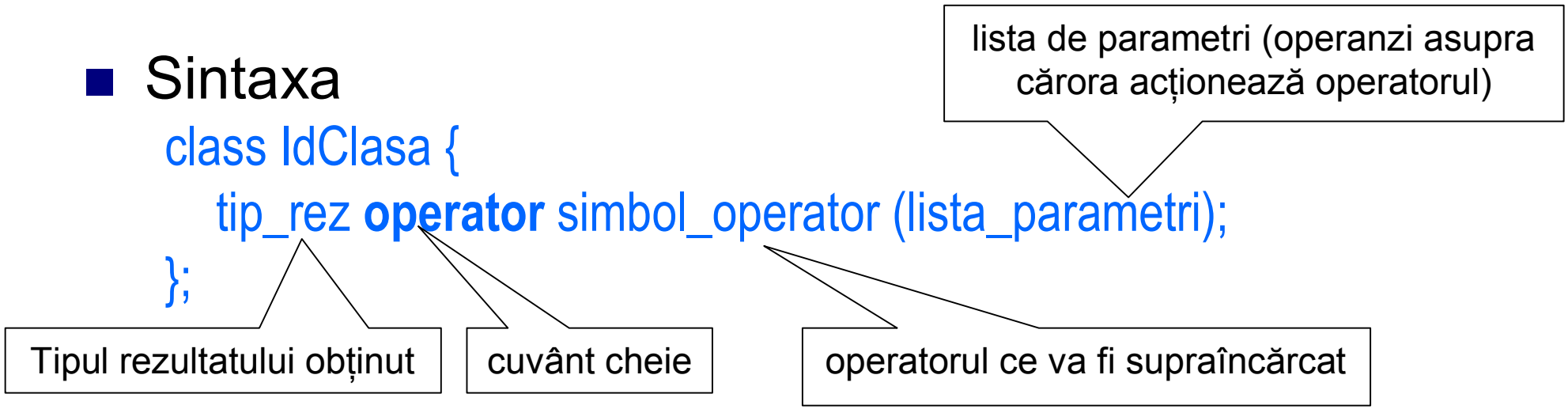
Supraîncărcarea operatorilor

- Operatori ce se supraîncarcă **obligatoriu** cu *funcție membru*:
  '`()`' (funcție), '`[]`' (indexare), '`->`' (calificare prin pointer) şi '`=`' (asignare).
- Operatori ce se supraîncarcă **de obicei** cu *funcție membru*:
  '`op=`' (operație cu atribuire), *operatorii unari*.
- Operatori ce se supraîncarcă **obligatoriu** cu *funcție membru statică sau cu funcție statică (globală)*:
  `new` şi **`delete`**
- *Ceilalți operatori se supraîncarcă cu funcție membru sau cu funcție (globală)*.

Supraîncărcarea operatorilor cu funcție membru

- Sintaxa

```
class IdClasa {
    tip_rez operator simbol_operator (lista_parametri);
};
```



- Numărul de parametri este cu 1 mai mic decât aritatea operatorului.
- Primul operand este obiectul curent pentru care se apelează operatorul.

Supraîncărcarea operatorilor cu funcții friend

- Sintaxa

```
class IdClasa {
  friend tip_rez operator simbol_operator(lista_parametri);
};

tip_rez operator simbol_operator (lista_parametri) {
}
```

- Numărul de parametri este egal cu aritatea operatorului.

Supraîncărcarea operatorilor. Exemplu

```
class Complex {
  private:
    float re;
    float im;

  public:
    Complex (float re = 0, float im = 0);
    void afisare();
    Complex operator +(Complex z);
    Complex operator ~();
    friend Complex operator -(Complex z1,
                              Complex z2);
    friend Complex operator -(Complex z);
};

Complex::Complex(float re, float im) {
   this->re = re;
   this->im = im;
}

void  Complex::afisare() {
    printf("%-g%+g*i\n", re, im);
}
```

```
Complex Complex::operator +(Complex z) {
   Complex rez;
   rez.re = this->re + z.re;
   rez.im = this->im + z.im;
   return rez;
}
Complex Complex::operator ~() {
   return Complex(re, -im);
}
Complex operator -(Complex z1, Complex z2) {
   return Complex(z1.re-z2.re, z1.im-z2.im);
}
Complex operator -(Complex z) {
   return Complex(-z.re, -z.im);
}

int main() {
  Complex z1(4,5), z2(3,1), z;
  z = z1 + z2; z.afisare();
  z = z1 - z2; z.afisare();
  z = ~z1; z.afisare();
  z = -z1; z.afisare();
  return 0;
}
```

Supraîncărcarea operatorilor ‘++’ şi ‘--’

Preincrementare (++a)

- cu funcție membru:

```
class IdClasa {
 IdClasa& operator ++ ();
};
```

- cu funcție prieten:

```
class IdClasa {
 friend IdClasa& operator ++
  (IdClasa &ob);
};
```

Postincrementare (a++)

- cu funcție membru:

```
class IdClasa {
 IdClasa& operator ++ (int n);
};
```

- cu funcție prieten:

```
class IdClasa {
 friend IdClasa& operator ++
   (IdClasa &ob, int n);
};
```

Supraîncărcarea operatorilor ‘++’ şi ‘--’

```
class Complex {
 private:
   float re;
   float im;

 public:
  Complex (float re = 0, float im = 0);
  void afisare();
  Complex& operator ++();
  Complex operator ++(int);
  friend Complex& operator --(Complex& z);
  friend Complex& operator --(Complex& z, int n);
};

Complex& Complex::operator ++() {
     re +=1;
     printf("preincrementare\n");
     return *this;
}

Complex Complex::operator ++(int n) {
     Complex c(*this);
     re +=1;
     printf("postincrementare\n");
     return c;
}
```

```
Complex& operator --(Complex& z) {
     z.re -=1;
     printf("predecremntare\n");
     return z;
}
Complex& operator --(Complex& z, int n) {
     z.re -=1;
     printf("postdecrementare\n");
     return z;
}
int main() {
  Complex z(4, 5);
  z.afisare();
  ++z;  z.afisare();
  z++;  z.afisare();
  --z;  z.afisare();
  z--;  z.afisare();
}
```

OUTPUT

```
4+5*i
preincrementare
5+5*i
postincrementare
6+5*i
predecremntare
5+5*i
postdecrementare
4+5*i
```

Constructorul de copiere

- *Constructorul de copiere* definit implicit de către compilator presupune copierea bit cu bit a elementelor.
- Constructorul de copiere se apelează la:
  - inițializarea obiectelor la definire;
  - inițializarea obiectelor transmise ca parametri prin valoare;
  - returnarea de obiecte la revenirea din funcții.
- Constructorul de copiere este obligatoriu să fie definit explicit pentru clasele ce conțin atribute alocate dinamic.

Supraîncărcarea operatorului ‘=‘

- Operatorul ‘=‘ se supraîncarcă numai cu funcție membru:

```
class IdClasa {
    IdClasa& operator = (const IdClasa&ob) ;
};

IdClasa& IdClasa::operator = (const IdClasa&ob){
    if (this != &ob){ // test pentru a evita atribuiri de tipul ob = ob
      // instructiuni de copiere a datelor membru
    }
    return *this;
}
```

Supraîncărcarea operatorului ‘=‘

- Dacă o clasă nu are supraîncărcat *operatorul egal (‘=‘)* atunci compilatorul va genera o supraîncărcare standard ce va realiza copierea **bit cu bit** a datelor membru.
- Dacă o clasă conține ca date membru, obiecte ale altor clase şi nu are supraîncărcat *operatorul ‘=‘*, atunci constructorul generat de compilator va apela la supraîncărcările operatorului ‘=‘ pentru copierea obiectelor membru.
- Se vor avea în vedere următoarele situații:

```
a = b;
a = b = c;
```

Supraîncărcarea operatorului ‘=‘. Exemplu

```
class Complex {
  private:
    float re;
    float im;
  public:
    Complex (float re = 0, float im = 0);
    Complex (const Complex &z);
    void afisare();

    Complex& operator = (const Complex& z);
};

Complex::Complex (float re, float im) {
   this->re = re;
   this->im = im;
}

Complex::Complex (const Complex &z) {
   this->re = z.re;
   this->im = z.im;
   printf("Apel constructor de copiere\n");
}
```

```
void  Complex::afisare() {
    printf("%-g%+g*i\n", re, im);
}

Complex& Complex::operator = (const Complex &z) {
   if (this != &z) {
     this->re = z.re;
     this->im = z.im;
   }
   printf("Apel supraincarcare =\n");
   return *this;
}

int main() {
  Complex z1(3,4);
  Complex z2 = z1; //Apel constructor de copiere
  z2.afisare();
  Complex z3;
  z3 = z1;         //Apel supraincarcare =
  z3.afisare();
  return 0;
}
```

Supraîncărcarea operatorului ‘=‘

- Corpul metodei va avea în vedere următoarele aspecte:
  - eliberarea memoriei obiectului destinație pentru membrii alocați dinamic;
  - alocarea de memorie pentru membrii alocați dinamic;
  - copierea conținutului membrilor obiectului sursă în membrii obiectului destinație.
- Dacă o clasă conține **variabile membru de tip pointer,** atunci trebuie implementate următoarele funcții:
  - Constructor,
  - Constructor de copiere,
  - Destructor,
  - Supraîncărcarea operatorului ‘=‘.

Supraîncărcarea operatorilor ‘<<’ şi ‘>>’

- Operatorii ‘<<‘ şi ‘>>’ se supraîncarcă pentru a putea insera date în fluxurile de ieşire şi extrage date din fluxurile de intrare.
- Biblioteca `iostream` suprascrie operațiile de scriere/citire pentru *tipurile implicite*.
- Pentru *noile tipuri de date definite de utilizatori* aceştia sunt responsabili cu suprascrierea lor.
- Nu pot fi supraîncărcați ca *funcții membre*, ci trebuie utilizate funcții non-membre, deoarece primul operand este un obiect de tipul *ostream/ istream* şi nu o referință la clasă.

Supraîncărcarea operatorilor ‘<<’ şi ‘>>’

- Se recomandă să nu introduceți mesaje şi formatare când supraîncărcați operatorii ‘<<‘ şi ‘>>’.

```
friend ostream& operator << (ostream&, const IdClasa&);
friend istream& operator >> (istream&, IdClasa&);

friend ofstream& operator << (ofstream&, const IdClasa&);
friend ifstream& operator >> (ifstream&, IdClasa&);

IdClasa a;
cin >> a;
cout << a;
```

Supraîncărcarea operatorilor relaționali

- *Operatorii relaționali* vor fi supraîncărcați ca funcții non-membre, deoarece primul operand ar putea fi o valoare de tip diferit față de clasă.
- Valorile operanzilor nu se modifică, deci ar trebui să fie *referințe constante*.
- Valoarea de return ar trebui să fie de tip `bool`.
- Dacă sunt specificați ca funcții membre, ele trebuie să fie constante.
- Format

```
bool operator simbol_op (const IdClasa&, const IdClasa&);
bool operator simbol_op (tip, const IdClasa&);
bool operator simbol_op (const IdClasa&) const;
```

Supraîncărcarea operatorului ‘[]‘

- Se recomandă ca supraîncărcarea operatorului ‘`[]`’ să se realizeze ca funcție membru, primul operand fiind un membru al clasei.
- Pentru consistență, al doilea operator ar trebui să fie de tip întreg (transmis prin valoare).
- Deoarece starea obiectului nu se modifică, funcția ar trebui să fie constantă.
- Valoarea de *return* ar trebui să fie de tipul tabloului de elemente conținut de clasă.
- Format

```
tip& operator[] (tip);
const tip& operator[] (tip) const;
```

Supraîncărcarea operatorului ‘[]‘

```
class Vector {
    private:
      int* a;
      int n;
    public:
     Vector(int n = 10) {
       a = new int [n];
     }

     int& operator [](int k) const {
        return a[k];
     }
     ...
};

int main() {
    Vector u(100), v;
    cout << "u[1] = ";
    cin >> u[1];
    return 0;
}
```

Conversii de tip

- *La evaluarea unei expresii* se efectuează conversii sistematice de la tipurile întregi scurte la tipul `int` sau `unsigned int`. Apoi, pentru fiecare operand, dacă operanzii au tipuri diferite, se efectuează conversia necesară pentru a obține ambii operanzi de acelaşi tip.
- *La atribuire* se efectuează conversia valorii atribuite la tipul operandului care primeşte valoarea.
- *La transferul parametrilor* se efectuează conversia valorii fiecărui parametru efectiv la tipul parametrului formal.

Conversii de tip

- Conversii posibile
  - de la **tip de bază** la **tip clasă**
  - de la **tip clasă** la **tip de bază**
  - de la **tip clasă** la **tip clasă.**
- Cum se pot realiza?
  - *supraîncărcarea operatorului de cast*
  - *prin intermediul constructorilor.*

Prin intermediul constructorilor

- Realizează conversia de la un tip de dată (predefinit sau altă clasă definită de utilizator) la clasa al cărui constructor este.
- Prototip:

```
X(tipDeData)
```

- Apelare:
  - Declarații de tipul:

```
X x = valoare;
```

  - Apelul funcțiilor dacă este nevoie de o conversie la tipul de date.

Operatorul de cast

- Sintaxă

```
operator tipDeData();
```

- Observații
  - *Operatorul* este întotdeauna funcție membru a clasei.
  - Tipul elementului *returnat* este implicit tipul operatorului.
  - Nu are parametri deoarece primeşte implicit adresa obiectului curent.

Conversii de tip

- Supraîncărcarea operatorului de cast nu poate realiza conversii *dintr-un tip fundamental în tip clasă*.
- *Operatorul de cast* este *unar*.
- Definirea constructorilor nu permite conversia *unui tip clasă la un tip de bază*.
  - Transformarea *obiect* -> *alt tip de date* (fundamental sau definit de utilizator) se realizează cu operatorul de cast supraîncărcat.
  - Transformarea *tip de date* (fundamental sau definit de utilizator) -> *obiect* se realizează prin intermediul constructorului cu un parametru.

Supraîncărcarea operatorului ‘()‘

- `tip operator () (param)`
- Supraîncărcarea operatorului ‘`()`’ se realizează numai cu funcție membru.
- Supraîncărcarea acestui operator pune la dispoziție posibilitatea utilizării obiectelor ca funcții (*functor*).

Clasa Polinom

```
class Polinom {
    int grad;
    int *coef;
  public:
    Polinom(int n);
    Polinom(Polinom& P);
    ~Polinom() { delete[] coef; };

    int operator[] (int i) const {
      return ((0<=i && i<=grad) ? coef[i] : 0);
    };
    Polinom& operator +(Polinom& P);

    friend Polinom& operator*(Polinom& P1,
                              Polinom& P2);
    friend istream& operator>> (istream& is,
                                Polinom& P);
    friend ostream& operator<< (ostream& os,
                                Polinom& P);
};

Polinom::Polinom(int n) {
  grad = n; coef = new int[n + 1];
  memset(coef, 0, sizeof(int) * (n + 1));
}
```

```
Polinom::Polinom(Polinom& P) {
  grad = P.grad;
  coef = new int[grad + 1];
  for (int i = 0; i <= grad; i++)
    coef[i] = P.coef[i];
}

int max(int x, int y) {
  return ((x > y) ? x : y);
}

Polinom& Polinom::operator +(Polinom& P) {
  Polinom *T;
  int i;

  T = new Polinom(max(grad, P.grad));
  for (i = 0; i <= T->grad; i++)
    T->coef[i] = (*this)[i] + P[i];

  for (i = T->grad; i >= 0 && !T->coef[i]; i--);
    T->grad = i;

  return *T;
}
```

Clasa Polinom

```
Polinom& operator *(Polinom& P1, Polinom& P2) {
  Polinom *T;

  T = new Polinom(P1.grad + P2.grad);
  for (int i = 0; i <= P1.grad; i++)
    for (int j = 0; j <= P2.grad; j++)
      T->coef[i + j] += P1[i] * P2[j];

  return *T;
}

istream& operator >> (istream& is, Polinom& P){
  for (int i = 0; i <= P.grad; i++) {
    cout << "coef[" << i << "]= ";
    is >> P.coef[i];
  }

  return is;
}

ostream& operator<< (ostream& os, Polinom& P) {
  os << "P(x)=";
  for (int i = P.grad; i >= 0; i--)
    if (P.coef[i]) {
      if (i != P.grad)
        os << ((P.coef[i] > 0) ? '+' : ' ');
      os << P.coef[i] << "*x^" << i;
    }
  os << endl;

  return os;
}

void main() {
  Polinom P(1), Q(2);

  cout << "Dati primul polinom: \n";
  cin >> P;
  cout << "Dati al doilea polinom: \n";
  cin >> Q;

  Polinom R = P + Q;
  cout << "Polinomul suma: \n";
  cout << R;
  R = P * Q;
  cout << "Polinomul produs: \n";
  cout << R;
}
```

Temă

- Implementați clasa *Matrice* reprezentată sub forma unui tablou unidimensional alocat dinamic. Supraîncărcați operatorii ‘+’ (suma), ’*’ (produs).