

Clase și funcții friend. Supraîncărcarea operatorilor

Mihai Gabrovanu

Funcții friend

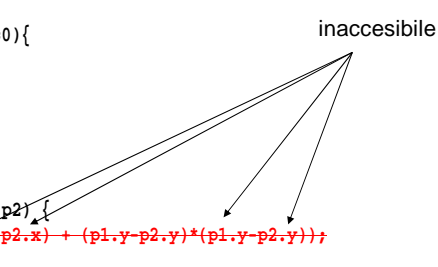
- n Funcțiile **friend** (prieten) sunt funcții asociate unor clase care au acces la datele și metodele protejate ale acelor clase deși nu sunt funcții membre ale acelei clase
- n Tipuri de funcții prieten
 - .. funcții globale
 - .. funcții membre ale altor clase

Clase si funcții friend. Supraîncărcarea operatorilor

3

Exemplu – Necesitatea accesării datelor protejate

```
class Punct {  
    private:  
        double x, y;  
    public:  
        Punct(double x=0, double y=0){  
            this -> x = x;  
            this -> y = y;  
        }  
};  
  
double distanta(Punct p1, Punct p2) {  
    return sqrt((p1.x-p2.x)*(p1.x-p2.x) + (p1.y-p2.y)*(p1.y-p2.y));  
}
```



Clase si funcții friend. Supraîncărcarea operatorilor

2

Funcții friend globale

- n Declararea unei funcții **friend globale pentru o clasă** se face incluzând prototipul ei, precedat de cuvântul cheie **friend**, în acea clasă

```
class IdClasa {  
    friend tip_ret id_functie_prieten(lista_de_parametri);  
};
```
- n Definiția funcției se face în afara clasei

```
tip_ret id_functie_prieten(lista_de_parametri) {  
    //corpul de instructiuni în care avem acces la datele/metodele protejate ale obiectelor clasei IdClasa  
}
```

Clase si funcții friend. Supraîncărcarea operatorilor

4

Funcții friend globale. Exemplu

```
class Punct {
private:
    double x, y;
public:
    Punct(double x=0, double y=0){
        this->x = x;
        this->y = y;
    }
    friend double distanta(Punct p1, Punct p2);
};
```

Funcția `distanta` este declarată
funcție prieten a clasei Punct

```
double distanta(Punct p1, Punct p2) {
    return sqrt((p1.x-p2.x)*(p1.x-p2.x) + (p1.y-p2.y)*(p1.y-p2.y));
}
```

```
int main(){
    Punct p1(1,0), p2(4,4);
    cout<<"distanta="<< distanta (p1,p2);
    getch();
}
```

Definiția funcției `distanta`.
Avem acces asupra datelor
private `x` și `y` din clasa Punct

Clase si funcții friend. Supraîncărcarea operatorilor

5

Exemplu: Funcții friend membre ale altor clase

```
class Punct;

class PolygonConvex{
    Punct *varfuri;
    int n;
public:
    PolygonConvex (int n, Punct v[]);
    ~PolygonConvex();
    void afiseaza();
};

class Punct {
private:
    double x, y;
public:
    Punct(double x=0, double y=0);
    friend void PolygonConvex::afiseaza();
};

Punct::Punct(double x, double y){
    this->x = x;
    this->y = y;
}
```

```
PolygonConvex::PolygonConvex(int n, Punct v[]){
    this->n = n;
    varfuri = new Punct[n];
    for (int i=0; i<n; i++){
        varfuri[i] = v[i];
    }
}

PolygonConvex::~PolygonConvex(){
    delete []varfuri;
}

void PolygonConvex::afiseaza() {
    cout<<"[";
    for (int i=0; i<n; i++){
        cout<<"("<< varfuri[i].x << " , "
        << varfuri[i].y<<")";
    }
    cout<<"]";
}

int main(){
    Punct t[3]={Punct(0,0), Punct(3,0),
    Punct(3,4)};
    PolygonConvex p(3,t);
    p.afiseaza();
}
```

Definiția funcției `afiseaza`.
Avem acces asupra datelor
private `x` și `y` din clasa Punct

Clase si funcții friend. Supraîncărcarea operatorilor

7

Funcții friend membre ale altor clase

- n Sunt funcții membre ale unei clase ce au acces la datele membru protejate ale unei alte clase.
- n Declararea unei funcții **friend** se face incluzând prototipul ei, precedat de cuvântul cheie **friend**, în clasa în care se dorește accesul

```
class IdClasaA; //declarare a clasei IdClasaA inaintea definirii clasei IdClasaB
class IdClasaB {
    tip_ret id_funcție_prieten(lista_de_parametri); //declararea funcției
};
```

```
class IdClasaA {
    friend tip_ret IdClasaB::id_funcție_prieten(lista_de_parametri);
};
```

```
tip_ret IdClasaB::id_funcție_prieten(lista_de_parametri) {
    //corpul de instrucțiuni în care avem acces la datele protejate ale obiectelor clasei IdClasaA
}
```

Clase si funcții friend. Supraîncărcarea operatorilor

6

Clase friend (I)

- n Dacă dorim ca **toate** metodele dintr-o clasă `IdClasaB` să aibe acces asupra tuturor datelor/funcțiilor membre protejate ale unei alte clase `IdClasaA` atunci declarăm clasa `IdClasaB` ca fiind clasă **friend** (prieten) pentru clasa `IdClasaA`
- n Sintaxa declarării claselor prietene este următoarea:

```
class IdClasaB; //declarare a clasei IdClasaB inaintea definirii clasei IdClasaA
class IdClasaA {
    friend class IdClasaB;
};
```

Clase si funcții friend. Supraîncărcarea operatorilor

8

Clase friend (II)

- n Relația de *prietenie* dintre două clase **nu este reflexivă**: Dacă clasa `IdClasaA` este clasă prieten a clasei `IdClasaB`, atunci nu și reciproca este valabilă
- n Relația de prietenie **nu este tranzitivă**: dacă clasa `IdClasaA` este clasă prietenă clasei `IdClasaB`, iar `IdClasaB` este clasă prietenă clasei `IdClasaC`, asta *nu implică* faptul că `IdClasaA` este clasă prietenă a clasei `IdClasaC`.
- n Relația de prietenie **nu se moștenește** în clasele *derivate*.

Supraîncărcarea operatorilor

Clase Friend. Exemplu

```
class Segment; //Declaraare clasa Segment
class Punct {
private:
    double x, y;
    void afisare();
public:
    Punct(double x=0, double y=0);
    //Declaram clasa Segment ca fiind
    //functie prietena a clasei Punct
    friend class Segment;
};

class Segment {
    Punct o;
    Punct v;
public:
    Segment(Punct o, Punct v);
    double lungime();
    void afisare();
};

Punct::Punct(double x, double y) {
    this->x = x;
    this->y = y;
}

void Punct::afisare() {
    cout<<"("<<x<<" "<<y<<"<<endl;
}

Segment::Segment(Punct o, Punct v) {
    this->o = o;
    this->v = v;
}

double Segment::lungime() {
    return sqrt((o.x-v.x)*(o.x-v.x) +
                (o.y-v.y)*(o.y-v.y));
}

void Segment::afisare() {
    cout<<"[";
    o.afisare();
    cout<<" ";
    v.afisare();
    cout<<"]<<endl;
}

int main() {
    Punct o(1,0), v(4,4);
    Segment s(o,v);
    s.afisare();
    cout<<"\nLungime ="<< s.lungime();
    getch();
}
```

Introducere

- n Un **tip de date** (predefinit) definește un set de valori și o mulțime de operații ce se pot efectua cu acestea. De exemplu: adunare (+), scădere (-), etc.
- n Problemă: în cazul tipurilor de date definite prin intermediul claselor am putea defini anumite operații cu ajutorul operatorilor?

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();
}
```

Supraîncărcarea operatorilor

- n Este procesul de atribuire a două sau mai multor operații aceluiași operator.
- n Se poate realiza prin
 - .. Funcții membru
 - .. Funcții friend globale

Alternativa...

- n O exprimare mai elegantă a operațiilor:

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

- n E posibilă în C++ prin **supraîncărcarea operatorilor**:

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

Supraîncărcarea operatorilor. Restricții

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

Supraîncărcarea operatorilor cu funcții membru

n Sintaxa

```
class IdClasa {
```

```
    tip_rez operator simbol_operator (lista_parametri) ;
```

```
};
```

Tipul rezultatului obținut

cuvânt cheie

operatorul ce va fi supraîncărcat

lista de parametri (operandi asupra
carora acționează operatorul)

n Numărul de parametri este cu 1 mai mic decât aritatea operatorului.

n Primul operand este obiectul curent pentru care se apelează operatorul

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("%f-%fi\n", re, im);
```

```
}
```

Supraîncărcare cu
funcții membru

```
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=z1 - z2;  
    z=-z1;  
    z=-z1
```

```
};
```

Supraîncărcare cu
funcții friend

Supraîncărcarea operatorilor cu funcții friend

n Sintaxa

```
class IdClasa {
```

```
    friend tip_rez operator simbol_operator (lista_parametri) ;
```

```
};
```

```
tip_rez operator simbol_operator (lista_parametri) {
```

```
}
```

n Numărul de parametri este egal cu aritatea operatorului.

Supraîncărcarea operatorilor ++ și --

Preincrementare (++a)

n cu funcție membru

```
class IdClasa {
```

```
    IdClasa& operator ++ () ;
```

```
};
```

n cu funcție prieten

```
class IdClasa {
```

```
    IdClasa& operator ++ (IdClasa &ob) ;
```

```
};
```

Postincrementare (a++)

n cu funcție membru

```
class IdClasa {
```

```
    IdClasa& operator ++ (int n) ;
```

```
};
```

n cu funcție prieten

```
class IdClasa {
```

```
    IdClasa& operator ++ (IdClasa &ob, int n) ;
```

```
};
```

Supraîncărcarea operatorilor ++ și --

```
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){
    re +=1;
    printf("postincrementare\n");
    return *this;
}
```

```
Complex& operator --(Complex& z){
    z.re -=1;
    printf("predecrementare\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
predecrementare
5+5*i
postdecrementare
4+5*i
```

Clase si functii friend. Supraîncărcarea operatorilor

21

Supraîncărcarea operatorului =

- n Dacă o clasă nu are supraîncărcat operatorul egal atunci compilatorul va genera automat o supraîncărcare standard care va realiza copierea **bit cu bit** a datelor membru
- n Dacă o clasă conține date membru obiecte ale altor clase și nu are supraîncărcat operatorul =, atunci constructorul generat de compilator va apela la supraîncărcările operatorului = pentru copierea obiectelor membru

Clase si functii friend. Supraîncărcarea operatorilor

23

Supraîncărcarea operatorului =

- n 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 atribuirii de tipul ob = ob
        //instrucțiuni de copiere a datelor membru
    }
    return *this;
}
```

Clase si functii friend. Supraîncărcarea operatorilor

22

Exemplu I: Supraîncărcarea operatorului =

```
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%+gi\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();
    getch();
}
```

Clase si functii friend. Supraîncărcarea operatorilor

24

Exemplu II: Supraîncărcarea operatorului =

```
class Persoana {
private:
    char *nume;
    int varsta;
public:
    Persoana(char *n="", int v=0){
        nume = new char[strlen(n)+1];
        strcpy(nume, n);
        varsta = v;
        cout<<"Apel constructor cu parametri\n";
    }

    Persoana(const Persoana &p){
        nume = new char[strlen(p.nume)+1];
        strcpy(nume, p.nume);
        varsta = p.varsta;
        cout<<"Apel constructor de copiere\n";
    }

    ~Persoana(){
        cout<<"Distrug obiectul:"<<nume<<endl;
        delete []nume;
    }

    void setNume(char *n){
        if(strlen(nume)<strlen(n)){
            delete nume;
            nume = new char[strlen(n)+1];
        }
        strcpy(nume, n);
    }

    void setVarsta(int v){
        varsta = v;
    }

    void afisare(){
        cout<<"Nume:"<<nume<<endl;
        cout<<"Varsta:"<<varsta<<endl;
    }
};
```

Clase si funcții friend. Supraîncărcarea operatorilor

25

Exemplu III: Supraîncărcarea operatorului =

```
class Persoana {
    ...
public:
    ...
    Persoana& operator =(const Persoana &ob){
        if (this != &ob){
            setNume(ob.nume);
            setVarsta(ob.varsta);
        }
        cout<<"Apel supraincercare = \n";
        return *this;
    }
};

int main(){
    Persoana p1("Mihai",21);
    Persoana p2=p1;
    Persoana p3;
    p3 = p1;
    p1.setNume("Misu");
    p1.afisare();
    p2.afisare();
    p3.afisare();
}
```

OUTPUT
Apel constructor cu parametri
Apel constructor de copiere
Apel constructor cu parametri
Apel supraincercare =
Nume:Misu
Varsta:21
Nume:Mihai
Varsta:21
Nume:Mihai
Varsta:21
Distrug obiectul:Mihai
Distrug obiectul:Mihai
Distrug obiectul:Misu

Clase si funcții friend. Supraîncărcarea operatorilor

27

Exemplu II: Supraîncărcarea operatorului =

```
int main(){
    Persoana p1("Mihai",21);
    Persoana p2=p1;
    Persoana p3;
    p3 = p1;
    p1.setNume("Misu");
    p1.afisare();
    p2.afisare();
    p3.afisare();
}
```

OUTPUT
Apel constructor cu parametri
Apel constructor de copiere
Apel constructor cu parametri
Nume:Misu
Varsta:21
Nume:Mihai
Varsta:21
Nume:Misu (Corect ar fi fost Mihai)
Varsta:21
Distrug obiectul:Misu
Distrug obiectul:Mihai
Distrug obiectul:R06

Clase si funcții friend. Supraîncărcarea operatorilor

26

Temă

- n Implementați clasa Polinom pentru care supraîncărcați operatori +, * pentru a efectua adunarea și înmulțirea a doi vectori
- n Implementați clasa Matrice reprezentată sub forma unui tablou unidimensional alocat dinamic. Supraîncărcați operatori + (suma), * (produs), = (atribuire), [] (accesul la elemente matricei).

Clase si funcții friend. Supraîncărcarea operatorilor

28