PROGRAMAREA ORIENTATĂ OBIECT C++

Conf.univ.dr. Ana Cristina DĂSCĂLESCU

Universitatea Titu Maiorescu

- Accesul la membrii protejați ai unei clase poate fi acordat unei funcții independente (nu este membră clasei), dacă acesta este declarată prin modificatorul **friend**.

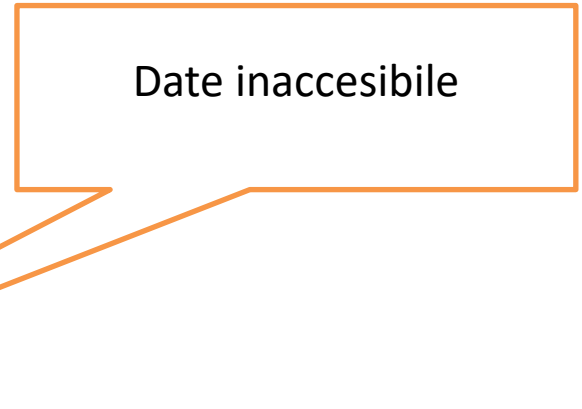
- Pentru a defini o funcție de de tip **friend**, se include prototipul ei în clasă, dar se definește în altă parte a programului astfel:

```
class X{
    ...
    friend tip_returnat  nume_functie(lista_argumente);
};
.....
tip_returnat  nume_functie (lista_argumente)
{...}
```

- **Functia friend** rămane externă, nefiind legată de clasă sau de un anumit obiect;
- Se declară în interiorul clasei, dar se definește în afara ei;
- Pentru a utiliza datele membre ale unui obiect, funcția friend primește ca parametru de intrare **referința către obiectul respectiv**;
- Relația de prietenie nu este simetricã si nici tranzitivă.

➢ **Exemplu**

```
class Punct {
  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));
}
```



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

Funcția **distanta** este declarată

funcție **prieten** a clasei Punct

Definiția funcției **distanta**.

Acces asupra datelor

- O funcție friend poate fi și o metodă membră a unei alte clase
- În acest caz, declararea funcției friend se face se realizează în clasa care oferă drepturi de acces.
  - Exemplu

```
class Medic
{…
 public:
    void consult(persoana &ob);
};
Class Persoana
{
    char nume[20];
    int varsta;
    friend void Medic::consult(Persoana &ob);
}
```

- Un **tip de date** defineşte un set de valori şi o mulţime de operaţii ce se pot efectua pe acesta. De exemplu, pentru un tip de date real (*float*, *double*) se pot aplica operatori aritmetici (+ ,- ,* , /), operatorul de incrementare (++), operatorul de decrementare (--) etc.

- **Problemă:** pentru *tipurile abstracte de date* (TAD) nu se pot utiliza operatorii predefiniţi pentru tipurile fundamentale din limbaj. De exemplu, pentru tipul abstract de date **Complex** nu se pot aplica implicit operatorii aritmetici!

- **O soluţie:** definirea unor metode membre/funcții friend care să implementeze operaţiile necesare.

➢ **Exemplu:** definirea unor metode membre/funcții friend pentru adunarea și scăderea a două obiecte de tip **Complex**

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

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

void Complex::afisare()
{
   cout<<re<<" "<<im;
}
```

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

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);
  return 0;
}
```

Alternativă

- Definirea operatorilor care să acționeze asupra obiectelor unei clase permite un mod mult mai convenabil de a manipula obiectele decât prin folosirea unor metode ale clasei.

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

- O funcţie care defineşte pentru o clasă o operaţie echivalentă operaţiei efectuate de un operator asupra unui tip predefinit este numită **funcţie operator.**
- Majoritatea operatorilor limbajului C++ pot fi supraîncărcaţi, şi anume:

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

Restricții privind supraîncărcarea operatorilor

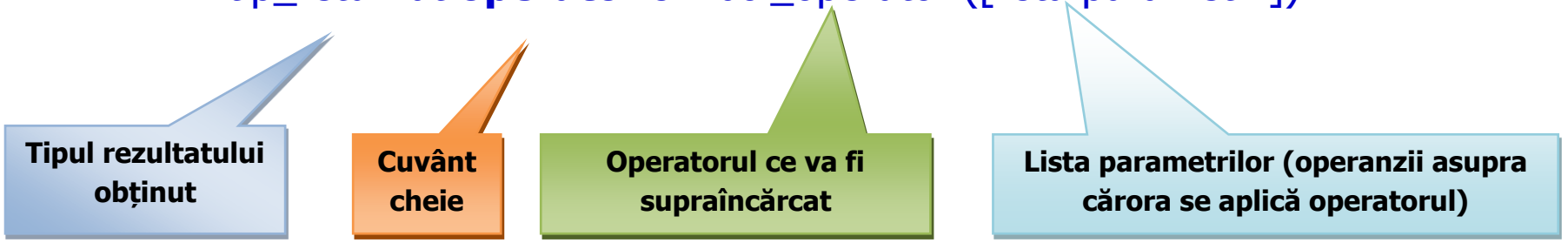
- Prin supraîncărcarea operatorilor **nu** se poate modifica:
  - **aritatea operatorilor**
  - **asociativiatea**
  - **prioritatea**
- Se pot supraîncărca numai operatori existenți în limbaj!
- Nu pot fi supraîncărcați următorii operatori:

`.`	`.*`	`::`	`?:`	`sizeof`

- Un operator se poate redefini prin funcții membre nestatice sau prin funcții externe unei clase.

- Forma generală a funcţiilor operator membre ale clasei este următoarea:

tip_returnat **operator** simbol_operator ([lista parametrii])



- O funcție membra de tip operator primește ca argument pointerul this.

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

- **Argumentul transmis funcţiei operator este operandul din dreapta operaţiei**, iar operandul din stânga este obiectul pentru care se apelează funcţia operator, adresat prin pointer *this*.

- **Sintaxa unei funcții operator de tip friend**:

```
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, fiind necesară transmiterea tuturor operanzilor, deoarece nu mai există un obiect al cărui pointer să fie transferat implicit funcţiei.

- Primul operand este obiectul curent pentru care se apelează operatorul.

- **Observație:** În multe cazuri, utilizarea fie a funcțiilor friend, fie a funcțiilor membre pentru supraîncărcarea unui operator nu provoacă diferențe funcționale programului. Totuși, în unele situații anumiți operatori se pot supraîncărca doar cu funcții membre, în timp ce alți operatori doar prin funcții friend.

- Pentru clasa **Complex**, cu datele membre `re` (partea reală) și `im` (partea imaginară), vom defini mai multe operații:
  - suma a două numere complexe, folosind o metodă membră;
  - conjugatul unui număr complex, folosind o metodă membră;
  - înmulțirea unui număr complex cu un scalar, folosind o funcție independentă de tip friend;
  - opusul unui număr complex, folosind o funcție independentă de tip friend.

SUPRAÎNCĂRCAREA OPERATORILOR - EXEMPLU

```
class  Complex {
 float  re,im;
 public:
   Complex(float re=0,float im=0);
   void afisare();
  //Supraîncărcare cu metode membru
  Complex operator+(Complex z);

  Complex operator~();
  //Supraîncărcare cu funcții friend
  friend Complex operator*(double v, Complex z2);

  friend Complex operator-(Complex z);
};
Complex::Complex(float re, float  im){
  this->re = re;
  this->im = im;
}
void Complex::afisare(){
   cout<<re<<" "<<im;
}
```

Funcţia supraîncarcă un operator binar. Argumentul transmis funcţiei este **operandul din dreapta operaţiei**. Adresa operandului din stânga este obiectul pentru care se apelează funcţia operator, accesat prin pointer-ul this.

Funcţia supraîncarcă un operator unar.

Funcţia supraîncarcă un operator binar. Deoarece funcția nu este membră a clasei, se transmit toți operanzii necesari.

SUPRAÎNCĂRCAREA OPERATORILOR - EXEMPLU

```
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*(double v, Complex z2){
 return Complex(z1.re*v,z1.im*v);
}

Complex operator -(Complex z){
 return Complex(-z.re,-z.im);
}

int main(){
 Complex z1(4,5), z2(3,1), z;
 z=z1+z2;
 z=3.5*z1;
 z=~z1;
 z=-z1;
 return 0;
}
```

OUTPUT

```
7.0 6.0
24.5 21.0
24.5 -21.0
-24.5 21.0
```

Echivalentă cu:

```
z = z1.operator+(z2);
```

Echivalentă cu:

```
z = operator(3.5,z1);
```

SUPRAÎNCĂRCAREA OPERATORILOR DE INCREMENTARE/DECREMENTARE

- Operatorii de incrementare ++ și decrementare -- sunt operatori unari care modifică operanzii.
- La supraîncărcarea operatorilor de incrementare sau decrementare (++, --) se poate diferenţia un operator prefixat de un operator postfixat folosind două versiuni ale funcţiei operator. Pentru varianta postfixată funcția operator++ sau operator -- are un argument suplimentar.

Operator de incrementare prefixat (++z)	Operator de incrementare postfixat (z++)

• **prin funcție membru:** se modifică obiectul current, deci se va returna adresa acestuia (pointer-ul *this*)
`class IdClasa{`
`  IdClasa&  operator ++ ();`
`};`
	• **prin funcție membru:** se modifică obiectul curent, deci se va returna adresa acestuia (pointer-ul this)
	`class IdClasa{`
	`  IdClasa& operator ++ (int n) ;`
	`};`

• **prin funcție de tip friend:** se modifică obiectul transmis ca parametru și se returnează
`class IdClasa {`
`  IdClasa operator ++ (IdClasa &ob) ;`
`};`
	• **prin funcție de tip friend:** se modifică obiectul transmis ca parametru și se returnează
	`class IdClasa {`
	`  IdClasa operator ++ (IdClasa &ob,`
	`  int n) ;`
	`}`

SUPRAÎNCĂRCAREA OPERATORILOR DE INCREMENTARE/DECREMENTARE

```
class  Complex {
  float  re,im;
   public:
      Complex(float re=0,float im=0);

       //supraîncărcare prin metode membre
       Complex operator++();
       Complex& operator++(int a);

      //supraîncărcare prin metode friend
       friend Complex operator--(Complex &ob);
       friend Complex operator--(Complex &ob, int a);
};

Complex::Complex(float re, float  im){
       this->re = re;
       this->im = im;
}
Complex::Complex operator++(){
 Complex crt = *this;
 this->++re;
 this->++im;
 return crt;
}
```

Operator de incrementare prefixat

Operator de incrementare postfixat

```
Complex&::Complex operator++(int a){
 this->re++;
 this->im++;
 return *this;
}
Complex operator--(Complex &ob){
 Complex crt = ob;
 this->--ob.re;
 this->--ob.im;
 return crt;
}
Complex operator--(Complex &ob,int a){
 this->ob.re--;
 this->ob.im--;
 return ob;
}

int main()
{
    Complex z1(1.2,3.2), z2;
    ++z1;//preincrementare
    z2++;//postincrementare
   return 0;
}
```

Observație:
Pentru funcţia `operator++(int a)` parametrul `a` are valoarea `0`.

SUPRAÎNCĂRCAREA OPERATORULUI DE ASIGNARE

- Pentru asignarea a două obiecte de același tip, în limbajul C++ se poate utiliza definiţia implicită de asignare, prin care se realizează copierea la nivel de bit a datelor membre ale obiectului sursă.

```
Complex z1(1,2), z2;
z2=z1; //copiere la nivel de bit
```

- **Problemă:** Dacă datele membre ale unei clase sunt alocate dinamic, copia bitwise, care se execută implicit la asignare, conduce la existența a doi pointeri care vor indica către aceeaşi zonă de memorie.

```
class X{
 public:
   char *p;

    X(char *p){
        this->p=new char[10];
        strcpy(this->p,p);
    }

    void afisare(){
        cout<<p<<" ";
    }
};
```

```
int main(){
    X ob1("abc"),ob2("def");
    ob1=ob2;
    ob1.afisare();
    strcpy(ob2.p,"aaaa");
    ob1.afisare();
    ob2.afisare();
    return 0;
}
```

Modificarea `ob2` conduce la modificarea `ob1`

```
Output: def  aaa aaa
```

➢Operatorul de asignare = se supraîncarcă numai prin funcție membru!

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

➢Pentru clasele cu date alocate dinamic, funcția **operator=** eliberează spațiul ocupat de obiectul pentru care se realizează asignarea, alocă un nou spațiu pentru obiectul care va fi copiat și realizează copia datelor membre ale obiectului sursă.

➢**Exemplu:** Definim clasa **String** care conţine un pointer `pstr` la un şir de caractere şi o variabilă de tip întreg `dim` care memorează dimensiunea vectorului de caractere corespunzător.

SUPRAÎNCĂRCAREA OPERATORULUI DE ASIGNARE

```
class String{

      char *pstr;
      int dim;
      public:
       String(const char *p);
       String(const String& r);
      ~String();
       String& operator=(const String &op2);
};

String::String(const char *p){
    dim = strlen(p) + 1;
    pstr = new char[dim];
    strcpy(pstr, p);
}

String::String(const String& r){
    dim = r.dim;
    pstr = new char[dim];
    strcpy(pstr, r.pstr);
}
```

Constructor de copiere

Supraîncărcarea operatorului de asignare

```
String::String(const String& r){
    dim = r.dim;
    pstr = new char[dim];
    strcpy(pstr, r.pstr);
}

String::~String(){
    if (pstr) delete []pstr;
    dim = 0;
}

String& String::operator=(const String &op2){
    if (str) delete []str;
    size = op2.size;
    str = new char[size];
    strcpy(str, op2.str);
    return *this;
}

int main(){
   String sir1("abc"), sir2("def");
   String sir3 = sir1;
   sir2 = sir3;
}
```

Apel constructor de copiere

Apel operator de asignare

- Supraîncărcarea funcţiilor şi a operatorilor (*overloading*) sunt mecanisme importante în C++ care oferă flexibilitate şi extensibilitate limbajului.

- Se pot supraîncărca doar operatorii existenți în limbaj.

- Nu se pot supraîncărca opeartorii: `.  .* :: ?: sizeof`

- Operatorii se pot supraîncărca prin metode membre sau prin funcții de tip friend.

- Sunt operatori care se pot supraîncărca doar prin metode membre (=, [] etc.) și operatori care se pot supraîncărca doar prin funcții friend (<<, >>).