PROGRAMAREA ORIENTATĂ OBIECT C++

Conf.univ.dr. Ana Cristina DĂSCĂLESCU

Universitatea Titu Maiorescu

➢Polimorfimul în limbajul C++

- Conceptul este o consecință a faptului ca orice obiect poate fi referit printr-un pointer de tipul său sau printr-un pointer de tipul clasei de bază, respectiv se poate realiza, într-un mod implicit, conversia unei clase derivate la o de bază, mecanism care poartă denumirea de **upcasting.**

- Considerăm clasa A extinsă de clasa B. Pot avea loc următoarele instanțieri:

```
B *b = new B();
```
//referirea obiectului printr-un pointer de tipul clasei

```
A *a = new B(...);
```
//referirea obiectului printr-un pointer de tipul clasei de bază

- Pentru obiectul **A *a = new B(...)**, se diferențiază două situații:

  - o metodă non virtuală, definită în clasa de bază și redefinită în clasa derivată, se apelează în raport cu tiplul pointer-ului, deci din clasa de bază!!!

  - o metodă virtuală, definită în clasa de bază și redefinită în clasa derivată, se apelează in raport cu tipul obiectului, deci din clasa derivată!!!

- **Intercorelarea (legarea) timpurie** se referă la evenimentele care se desfăşoară în timpul compilării, respectiv apeluri de funcţii pentru care sunt cunoscute adresele de apel:
  - funcţii membre non virtaule
  - funcţii supraîncărcate
  - operatori supraîncărcaţi,
  - funcţii friend

- **Polimorfismul introdus prin mecanismul de virtualitate** este polimorfism la nivel de executare, care permite legarea târzie (late binding) între evenimentele din program.
  - În astfel de apeluri, adresa funcţiei care urmează să fie apelată nu este cunoscută decât în momentul executării programului.

- **Funcţiile virtuale sunt evenimente cu legare târzie**: accesul la astfel de funcţii se face prin pointer la clasa de bază, iar apelarea efectivă a funcţiei este determinată în timpul execuţiei de tipul obiectului indicat, pentru care este cunoscut pointerul la clasa sa de bază.

- Avantajul principal al legării târzii (permisă de polimorfismul asigurat de funcţiile virtuale) îl constitue simplitatea şi flexibilitatea programelor rezultate.

- Observație
  - Apelurile rezolvate în timpul compilării beneficiază de o eficienţă ridicată, spre deosebire de apleurile efectuate la executare!!!

➢ Funcții virtuale pure

- De cele mai multe ori, o funcţie declarată de tip virtual în clasa de bază nu defineşte o acţiune semnificativă, dar este necesar ca ea să fie redefinită în fiecare din clasele derivate.
- Pentru ca programatorul să fie obligat să redefinească o funcţie virtuală în toate clasele derivate în care este folosită această funcţie, se declară funcţia respectivă **virtuală pură.**

- O funcţie virtuală pură este o funcţie care nu are definiţie în clasa de bază, ci se declară astfel:

```
virtual tip_returnat nume_functie(lista_argumente) = 0;
```

➢ Clase abstracte

- O clasă care conţine cel puţin o funcţie virtuală pură se numeşte **clasă abstractă**.
- Deoarece o clasă abstractă conţine una sau mai multe virtuale pure, nu pot fi create instanţe (obiecte), dar pot fi creaţi pointeri şi referinţe la astfel de clase abstracte.
- O clasă abstractă este folosită în general ca o clasă fundamentală, din care se construiesc alte clase prin derivare.
- Orice clasă derivată dintr-o clasă abstractă este, la rândul ei clasă abstractă (şi deci nu se pot crea instanţe ale acesteia) dacă nu se redefinesc toate funcţiile virtuale pure moştenite.
- Dacă o clasă redefineşte toate funcţiile virtuale pure ale claselor ei de bază, devine clasă ne-abstractă şi pot fi create instanţe (obiecte) ale acesteia.

Exemplu:

```
class AngajatUTM
{
protected:
    char nume[100];double salariu;
public:
    Persoana(char *nume, double salariu)
    {
        strcpy(this->nume, nume);
        this->salariu = salariu;
    }
   virtual double salariuSpor()=0;
};
```

```
class CadreDidactice:public AngajatUTM
{
public:
    Student(char *nume, double salariu)
                :AngajatUTM(nume, salariu)
    {

    }
   virtual double salariuSpor()
  {
     return salariu*1.1;
  }
};
```

```
class TESA:public AngajatUTM
{
public:
    Student(char *nume, double salariu)
                :AngajatUTM(nume, salariu)
    {

    }
   virtual double salariuSpor()
  {
     return salariu*1.15;
  }
};
```

```
int main()
{
AngajatUTM ob1;

AngajatUTM *ob1;
 int cod; cin>>cod;
    if(cod==1)
    {
        ob1  = new CadreDidactice();
        ob1->setSalariu(5000);
    }else
    {
        ob1 = new TESA();
        ob1->setSalariu(5000)
    }
   cout<<ob1->salariuSpor();

    return 0;}}
```

Eroare, o clasă abstractă nu se poate instanțoa

Apel din clasa derivată

- **Funcțiile și clasle template** reprezintă, alături de conceptul de moștenire, un mecanism de reutilizarea a unui cod creat anterior.

- Pentru o funcție template se specifică numărul argumentelor, împreună cu tipul lor generic.

- Sintaxa:

```
template<class X, class Y,…>
tip_returnat numeFunctie(X arg1, Y arg 2)
```

- Exemplu:

```
template<class Tip>
void interschimba(Tip &a, Tip &b)
    {
      Tip aux;
      aux = a;
       a = b;
       b = aux;
    }
```

```
int main()
{
   double x=1.1, y= 2.2;
   interschimba (x, y);
   int  a=1, b=2;
   interschimba (a, b);
   Persoana p1, p2;
   interschimba (p1, p2);
}
```

➢ Clase template

- Sunt descrieri parametrizate de clasă, ce vor fi adaptate ulterior diferitelor tipuri de date recunoscute în limbaj.

- O clasă template (generică) permite obținerea unor clase, particularizând tipul membrilor

- Sintaxa generală

```
template<class T1, ..., class Ti>
class IdClasa {T1 d1; T2 d2}
```

- Exemplu: Se definește o clasă generică pentru un obiect de tip vector alocat dinamic. Tipul elementelor stocate în vector este generic. Se solicită particularizări pentru diferite tipuri de dată, precum int, double, Persoana.

```
class Vector
{
    T* pe;
    int dim;
    public:
        Vector(int n);
        T& operator()(int i);
        void afisare();
        void sort();
};
```

```
template<class T> Vector<T>::Vector(int n)
{
    dim=n;
    pe= new T[n];

    for(int i=0; i<dim; i++)
        cin>>pe[i];
}
template<class T> T& Vector<T>::operator()(int i)
{
    return pe[i];}
```

```
template<classT>void Vector<T>::sort()
{
    int i, j; T aux;
    for(i = 0; i<dim-1; i++)
      for(j=i+1; j<dim;  j++)
        if(pe[i]>pe[j])
         {aux = pe[i];
           pe[i] = pe[j];
           pe[j] = aux;
}
```

```
Vector<int> vi(3);
vi.afisare();
vi.sort();
vi.afisare();



Vector<Persoana> vp(2);
vp.afisare();
vi.sort();
vp.afisare();
```

- Un șablon de clasă poate fi instanțiat devenind o clasă concretă prin substituirea tipurilor generice cu tipuri concrete.

- Deoarece nu se pot defini două clase cu același nume, programatorul poate referi clasele construite printr-o clasă template, printr-un nume compus din numele său, urmat de tipul de dată concret (tipul ce se specifică la instanțierea șablonului)
- Exemplu:
  `vector<int>, vector<double>, vector<Persoana>`.