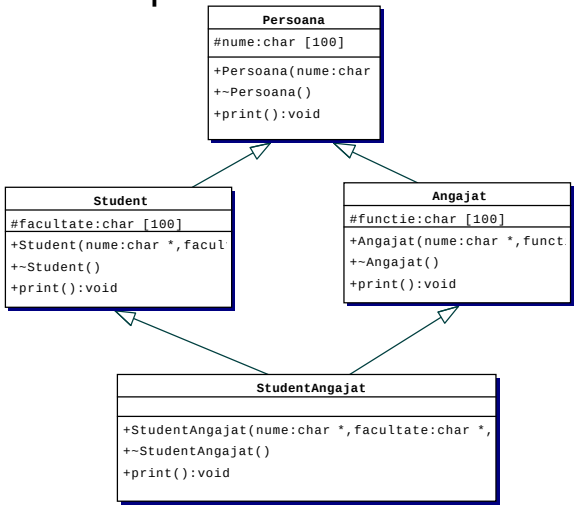
Moștenirea Multiplă. Clase Virtuale

Mihai Gabroveanu

Derivarea Multiplă. Problema Diamantului (I)



Derivarea Multiplă. Problema Diamantului (II)

```
class Persoana{
  protected:
    char nume[100];
  public:
    Persoana(char *nume){
      strcpy(this->nume, nume);
      cout<<"Apel constructor Persoana\n";
    }

    ~Persoana(){
      cout<<"Apel destructor Persoana\n";
    }
    void print(){
      cout<<"Nume:"<<nume<<endl;
    }
};
```

```
class Student: public Persoana{
  protected:
    char facultate[100];
  public:
  Student(char *nume,char *facultate):Persoana(nume){
      strcpy(this->facultate, facultate);
      cout<<"Apel constructor Student\n";
   }
   ~Student(){
     cout<<"Apel destructor Student\n";
    }
    void print(){
       Persoana::print();
       cout<<"Facultate:"<<facultate<<endl;
    }
};
```


Derivarea Multiplă. Problema Diamantului (II)

```
class Angajat: public Persoana{
  protected:
   char functie[100];
  public:
  Angajat(char *nume, char *functie):
                        Persoana(nume){
    strcpy(this->functie, functie);
    cout<<"Apel constructor Angajat\n";
  }
  ~Angajat(){
   cout<<"Apel destructor Angajat\n";
   }
   void print(){
    Persoana::print();
    cout<<"Functie:"<<functie<<endl;
   }
};
```

```
class StudentAngajat: public Student,
                      public Angajat{
 public:
  StudentAngajat(char *nume, char *facultate,
char *functie):
            Student(nume,facultate),
Angajat(nume, functie){
    cout<<"Apel constructor StudentAngajat\n";
  }

  ~StudentAngajat(){
     cout<<"Apel destructor StudentAngajat\n";
   }
   void print();
};
```

Derivarea Multiplă. Problema Diamantului (III)

```
StudentAngajat:: print() {
   cout<<"Nume: " <<nume<<endl;
   Student::print();
   cout<<"Functie: " <<functie<<endl;

}


int main(){
 StudentAngajat sa("Mihai", "Informatica",
"programator");
 sa.print();
 sa.Student::print();
 sa.Angajat::print();

 return 0;
}
```

```
Apel constructor Persoana
Apel constructor Student
Apel constructor Persoana
Apel constructor Angajat
Apel constructor StudentAngajat
Nume:Mihai
Facultate:Informatica
Functie:programator
Nume:Mihai
Facultate:Informatica
Nume:Mihai
Functie:programator
Apel destructor StudentAngajat
Apel destructor Angajat
Apel destructor Persoana
Apel destructor Student
Apel destructor Persoana
```


Probleme

- Duplicarea datelor membru din clasa Persoana la nivelul clasei StudentAngajat
- La nivelul clasei StudentAngajat nu putem să referim direct data membru nume moștenită de la clasele de bază.
- Putem însă folosi operatorul de rezoluție pentru a referi această dată membru

  Student::nume respectiv Angajat::nume

Clase Virtuale

- Pentru a evita duplicarea datelor membru, în cazul moștenirilor multiple, clasa de bază se declară **virtuală** în declarațiile claselor derivate:

```
class B { protected tip x; };
class B1: virtual public B { };
class B2: virtual public B { };
class D: public B1, public B2 { };
```

- Rezultat: data membru **x** se include o singură dată la nivelul clasei derivate **D**.

Clase virtuale. Constructori

- Constructorul clasei derivate D trebuie să apeleze explicit apelul clasei B în vederea creării copiei unice a datelor moștenite de la clasa B, pe lângă apelul constructorilor claselor de bază:

```
class D: public B1, public B2 {
  D(...): B(...), B1(...), B2(...){
   ...
  }
};
```

- Ordinea de apel a constructorilor este B, B1, B2,D

Exemplu

```
class Student: virtual public Persoana{
...
};

class Angajat: virtual public Persoana{
...
};

class StudentAngajat: public Student, public Angajat{
 public:
  StudentAngajat(char *nume, char *facultate, char *functie):
              Angajat(nume, functie), Student(nume,facultate), Persoana(nume){
    cout<<"Apel constructor StudentAngajat\n";
  }

  ~StudentAngajat(){
     cout<<"Apel destructor StudentAngajat\n";
   }
   void print(){
     cout<<"Nume: " <<nume<<endl;
     cout<<"Facultate:"<<facultate<<endl;
     cout<<"Functie:"<<functie<<endl;
    }
};
```


Exemplu (cont.)

```
int main(){
 StudentAngajat sa("Mihai", "Informatica",
"programator");
 sa.print();
 sa.Student::print();
 sa.Angajat::print();

 return 0;
}
```

```
Apel constructor Persoana
Apel constructor Student
Apel constructor Angajat
Apel constructor StudentAngajat
Nume:Mihai
Facultate:Informatica
Functie:programator
Nume:Mihai
Facultate:Informatica
Nume:Mihai
Functie:programator
Apel destructor StudentAngajat
Apel destructor Angajat
Apel destructor Student
Apel destructor Persoana
```


Constructorul de copiere în procesul de moștenire

- Constructorii sunt funcții membre ce nu se moștenesc (deci implicit nici constructori de copiere)
- Considerăm următoarea clasă derivată

```
class D: public B1, public B2, ..., public Bn{
  ...
};
```


Constructorul de copiere în procesul de moștenire (cont.)

- Dacă clasa derivată D nu are definit constructor de copiere atunci compilatorul va genera unul automat care va apela constructori de copiere ai claselor de bază B1, …, Bn, iar datele specifice clasei D se vor copia bit cu bit

Constructorul de copiere în procesul de moștenire (cont.)

- Dacă clasa derivată D are definit constructor de copiere atunci acesta se va ocupa atât de copierea datelor membre moștenite de la clasele de bază B1, …, Bn (poate apela explicit constructori acestora), cât și de datele specifice clasei D

```
class D: public B1, public B2, ..., public Bn{
  . . .
  D (const D &ob): B1(ob), B2(ob),..., Bn(ob) {
    //copierea datelor specifice lui D din ob
  }
};
```


Supraîncărcarea operatorului =

- Supraîncărcarea operatorului = nu se moștenește
- Dacă clasa derivată nu supraîncarcă operatorul **=** atunci compilatorul va apela automat *operator=* al claselor de bază B1, …, Bn, iar datele specifice clasei D se vor copia bit cu bit

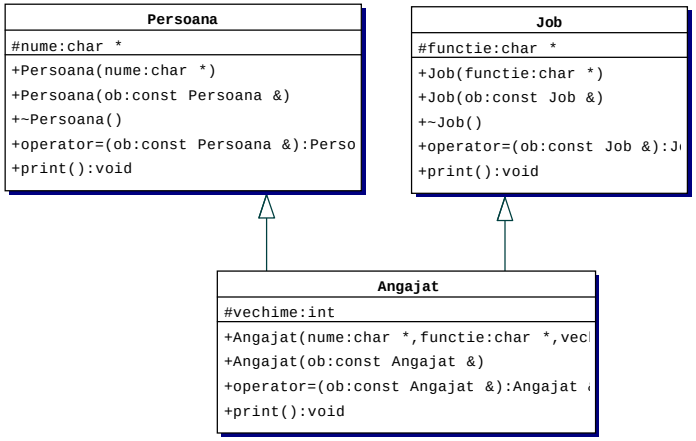
Supraîncărcarea operatorului =

- Dacă clasa derivată supraîncărcă operatorul **=** atunci el trebuie să realizeze copierea atât a datelor membre moștenite de la clasele de bază B1, …, Bn (poate apela explicit *operator=* al claselor de bază), cât și datele specifice clasei D

```
class D: public B1, public B2, ..., public Bn{
  . . .
  D & operator= (const D &ob) {
    if (this!=&ob){
    //copierea datelor specifice lui D din ob
     B1::operator =(ob);
     ...
     Bn::operator =(ob);
    }

    return *this;
  }
};
```


Exemplu



Exemplu(cont.)

```
class Persoana {
protected:
  char * nume;

public:
  Persoana(char * nume) {
    this->nume = new char[strlen(nume) + 1];
    strcpy(this->nume, nume);
    cout << "Apel constructor Persoana\n";
  }

  Persoana(const Persoana & ob) {
    this->nume = new char[strlen(ob.nume) + 1];
    strcpy(this->nume, ob.nume);
    cout << "Apel constructor copiere
Persoana\n";
  }
```

```
  ~Persoana() {
    delete[] nume;
    cout << "Apel destructor Persoana\n";
  }

  Persoana & operator = (const Persoana & ob) {
    if (this != & ob) {
      delete[] nume;
      this->nume=new char[strlen(ob.nume) + 1];
      strcpy(this->nume, ob.nume);
    }
    cout << "Apel operator= Persoana\n";
    return * this;
  }

  void print() {
    cout << "Nume:" << nume << endl;
  }
};
```


Exemplu(cont.)

```
class Job {
protected:
  char * functie;

public:
  Job(char * functie) {
   this->functie=new char[strlen(functie) + 1];
    strcpy(this->functie, functie);
    cout << "Apel constructor Job\n";
  }

 Job(const Job & ob) {
  this->functie=new char[strlen(ob.functie)+1];
  strcpy(this->functie, ob.functie);
  cout << "Apel constructor copiere Job\n";
 }
```

```
 ~Job() {
    delete[] functie;
    cout << "Apel destructor Job\n";
 }
 Job & operator = (const Job & ob) {
  if (this != & ob) {
   delete[] functie;
   this->functie=new char[strlen(ob.functie)+1];
   strcpy(this->functie, ob.functie);
  }
  cout << "Apel operator= Job\n";
  return * this;
 }

 void print() {
    cout << "functie:" << functie << endl;
 }
};
```


Exemplu(cont.)

```
class Angajat : public Persoana, public Job {
protected:
  int vechime;

public:
  Angajat(char * nume, char * functie, int
vechime = 0):
              Persoana(nume), Job(functie) {
    this->vechime = vechime;
    cout << "Apel constructor Angajat\n";
  }
  Angajat(const Angajat & ob) : Job(ob),
                                Persoana(ob) {
    vechime = ob.vechime;
    cout << "Apel constructor copiere
Angajat\n";
  }
```

```
  Angajat & operator = (const Angajat & ob) {
    if (this != & ob) {
      this->vechime = vechime;
      Persoana::operator = (ob);
      Job::operator = (ob);
    }
    cout << "Apel operator= Angajat\n";
    return * this;
  }

  void print() {
    Persoana::print();
    Job::print();
    cout << "Vechime:" << vechime << endl;
  }
};
```


Exemplu(cont.)

```
int main() {
  Angajat a1("Mihai", "programator", 1);
  a1.print();
  Angajat a2 = a1;
  a2.print();
  a1 = a2;
  a1.print();
  getch();
  return 0;
}
```

```
Apel constructor Persoana
Apel constructor Job
Apel constructor Angajat
Nume:Mihai
functie:programator
Vechime:1
Apel constructor copiere Persoana
Apel constructor copiere Job
Apel constructor copiere Angajat
Nume:Mihai
functie:programator
Vechime:1
Apel operator= Persoana
Apel operator= Job
Apel operator= Angajat
Nume:Mihai
functie:programator
Vechime:1
Apel destructor Job
Apel destructor Persoana
Apel destructor Job
Apel destructor Persoana
```