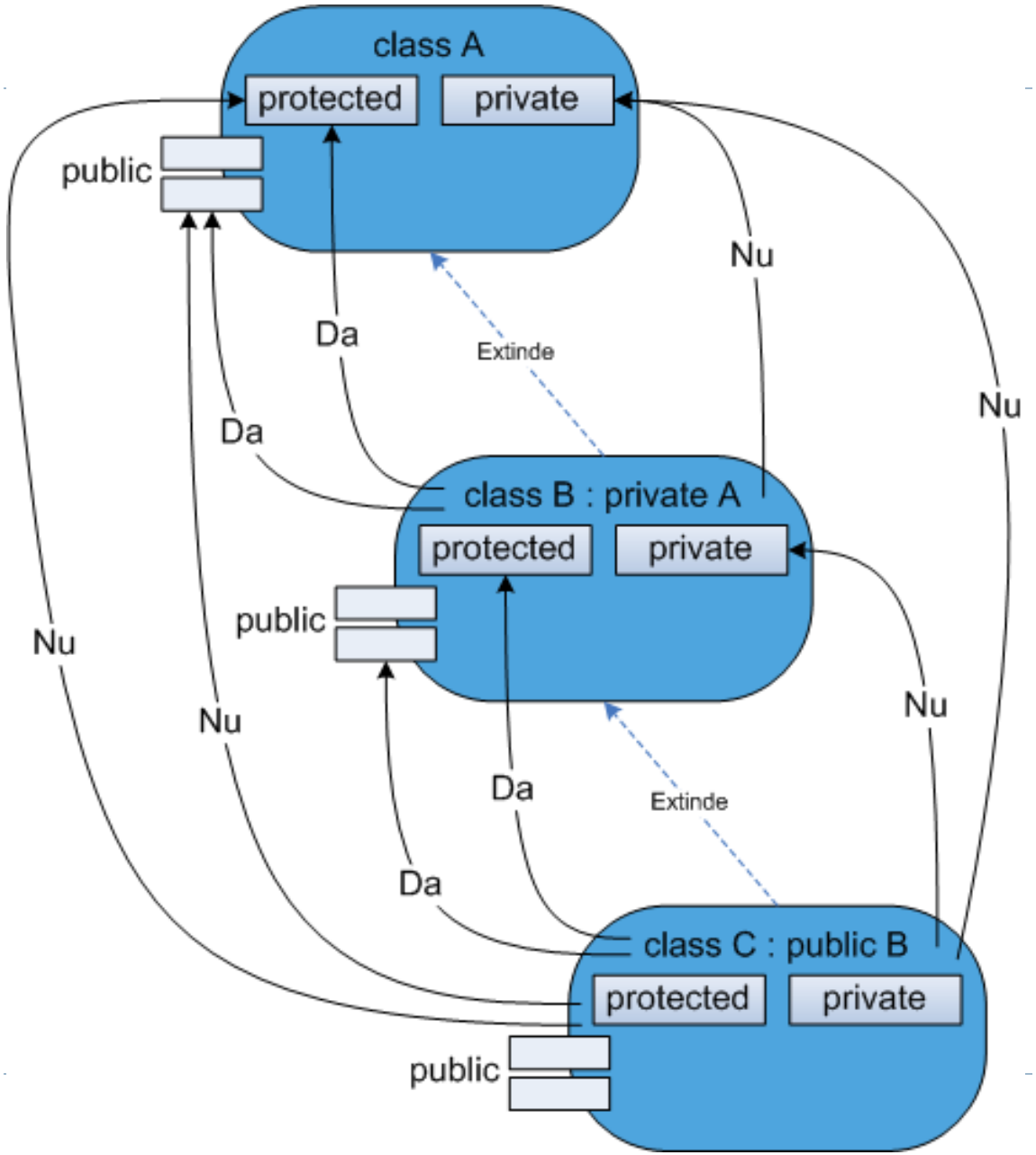
PROGRAMARE ORIENTATĂ PE OBIECTE

Curs 7

Moştenire - continuare

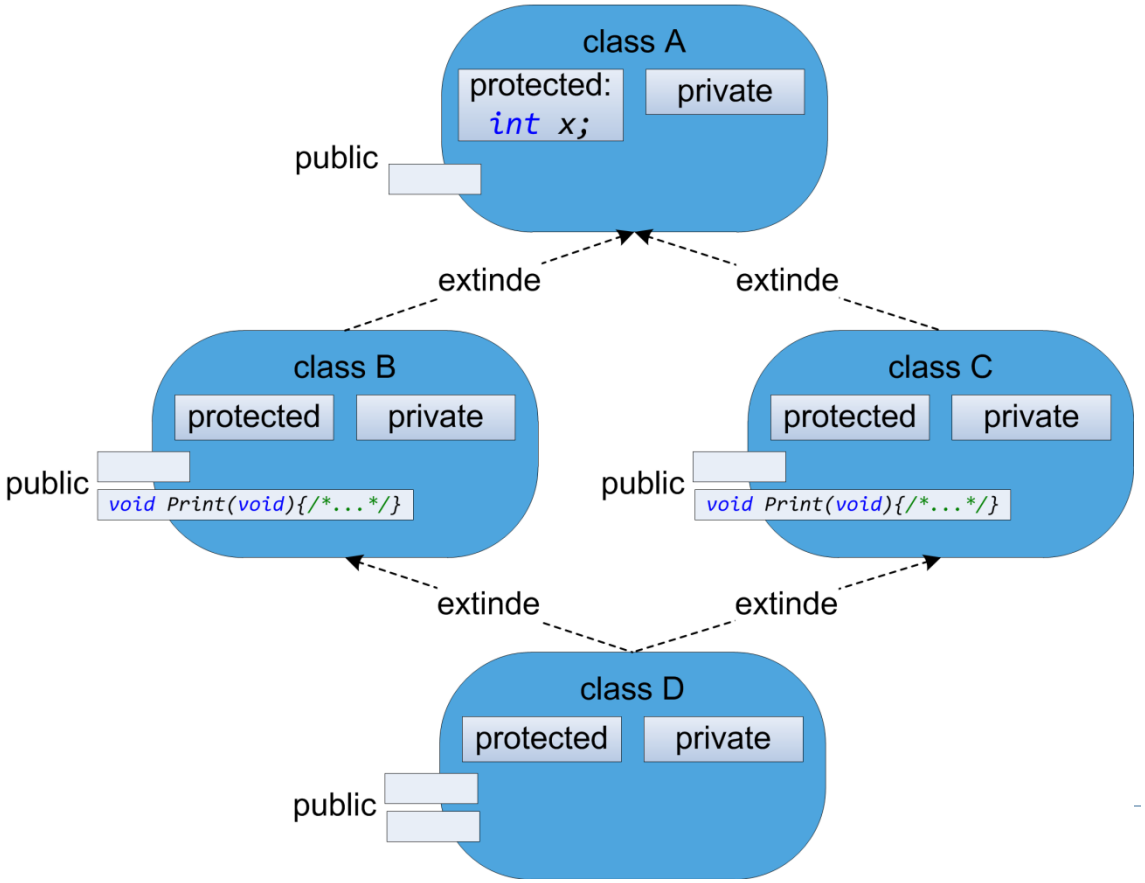
Moştenire multiplă – probleme
Polimorfism

Controlul accesului la membri clasei de bază



Moştenirea multiplă - probleme

- Moștenire în formă de diamant
- O clasă moștenește 2 clase care au o funcție membră cu același semnătură



Moştenirea multiplă – studiu de caz
Problema diamantului

- Fie următoarea ierarhie de clase

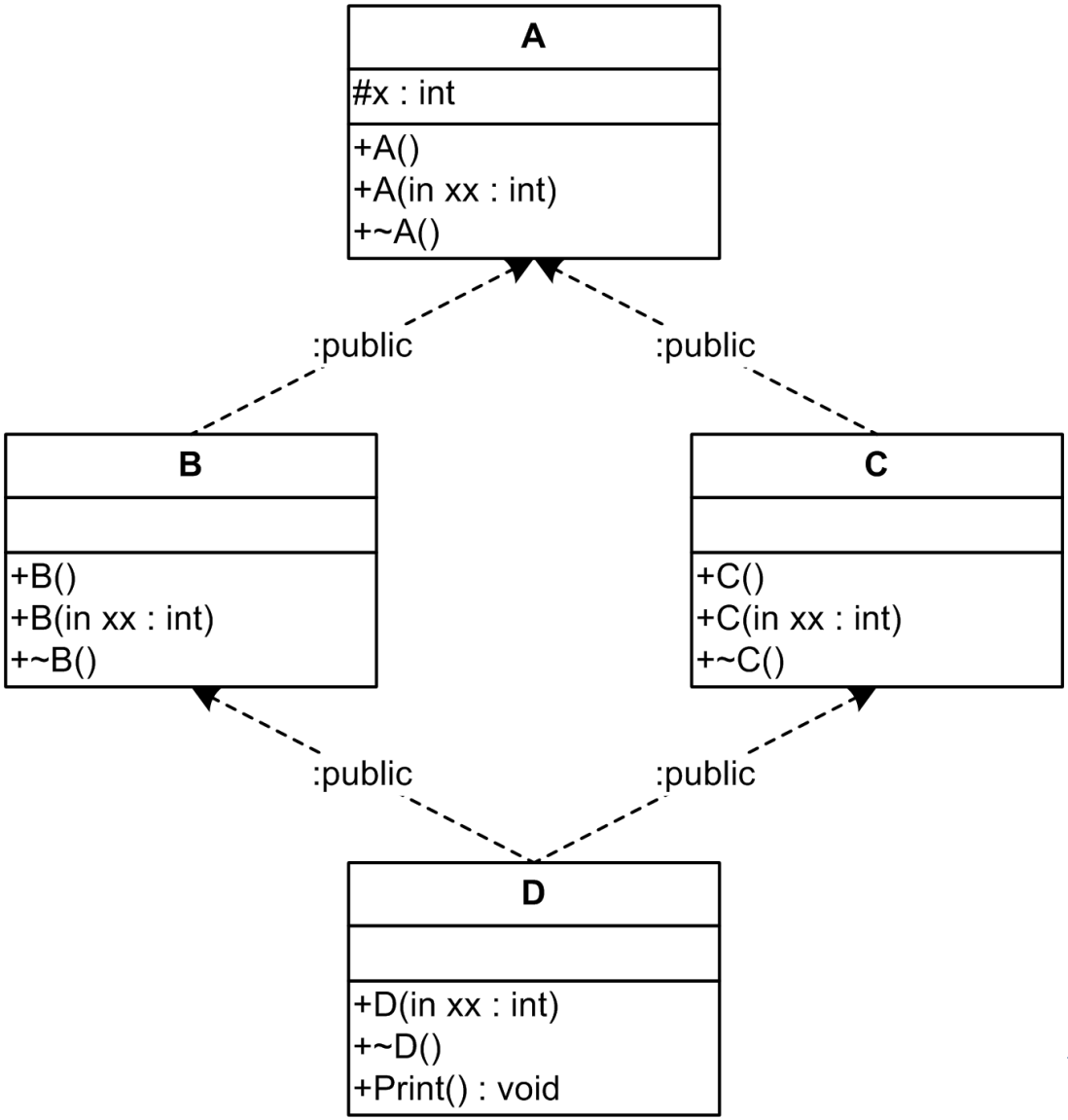
```
class A
{
protected:
    int x;
public:
    A(void) {x = 0; cout << "A()\n";}
    A(int xx) {x = xx; cout << "A(int)\n";}
    ~A(void) {cout << "~A()\n";}
};
class B: public A
{
public:
        B(void){cout << "B()\n";}
        B(int xx): A(x) {cout << "B(int)\n";}
        ~B(void) {cout << "~B()" << endl;}
};
```

Moştenirea multiplă – studiu de caz
Problema diamantului

```
class C: public A
{
public:
        C(void){ cout << "C()\n";}
        C(int xx) : A(xx) {cout << "C(int)\n";}
        ~C(void) {cout << "~C()\n";}
};

class D: public B, public C
{
public:
        D(int xx): B(xx), C() {cout << "D(int)\n";}
        ~D(void) {cout << "~D()\n";}
        void Print(void) {cout << x <<endl;};
};
```

Moştenirea multiplă – studiu de caz
Problema diamantului



Moştenirea multiplă – studiu de caz
Problema diamantului

```
int main (void)
{
        A a(6);B b(7);C c(8);D d(5);
        d.Print();
        cout <<"sizeof(a)="<<sizeof(a)<<endl;
        cout <<"sizeof(b)="<<sizeof(b)<<endl;
        cout <<"sizeof(c)="<<sizeof(c)<<endl;
        cout <<"sizeof(d)="<<sizeof(d)<<endl;
        return 0;
}
```

- Compilare:

```
error C2385: ambiguous access of 'x' in 'D'
```

- Posibile „soluții” de implementare ale funcției *D::Print()*:

```
void D::Print(void) {cout << A::x <<endl;}; //se afiseaza 5
void D::Print(void) {cout << B::x <<endl;}; //se afiseaza 5
void D::Print(void) {cout << C::x <<endl;}; //se afiseaza 0
```

Moştenirea multiplă – studiu de caz
Problema diamantului

- Dimensiuni ale obiectelor:

```
sizeof(a) = 4
sizeof(b) = 4
sizeof(c) = 4
sizeof(d) = 8
```

- Apelarea constructorilor:

```
A(int)
B(int)
A()
C()
D(int)
0
sizeof(d)=8
~D()
~C()
~A()
~B()
~A()
```

Moştenirea multiplă – studiu de caz
Problema diamantului

- Pentru rezolvarea cazului de ambiguitate se declara clasa A virtuala astfel încât constructorul clasei virtuale va fi apelat prima dată și apoi, celelalte apeluri explicite ale constructorului clasei virtuale vor fi ignorate

```
class B: public virtual A
{
public:
    B(void){cout << "B()\n";}
    B(int x): A(x) {cout << "B(int)\n";}
    ~B(void) {cout << "~B()";}
};
class C: virtual public A
{
public:
        C(void){cout << "C()\n";}
        C(int x): A(x) {cout << "C(int)\n";}
        ~C(void) {cout << "~C()\n";}
};
```

Moştenirea multiplă – studiu de caz
Problema diamantului

- Apel main:

```
A()
B(int)
C()
D(int)
D::x=0
sizeof(d)=16
~D()
~C()
~B()
~A()
```

- Pentru a se forța apelarea constructorului de initializare, se reimplementează constructorul de inițializare al clasei D.

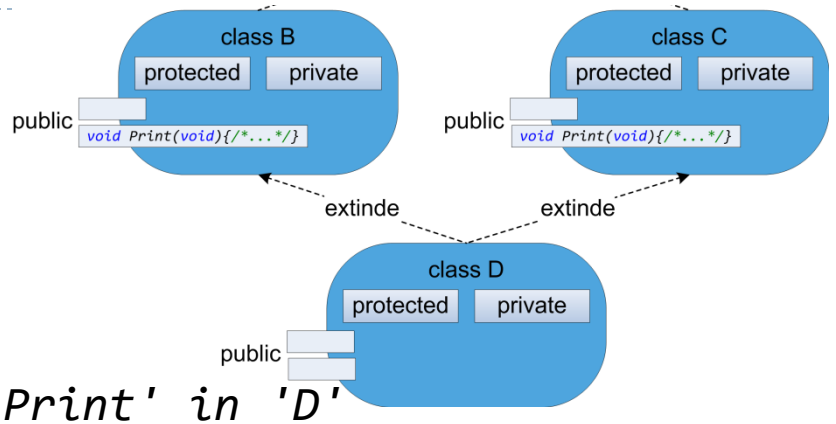
```
class D: public B, public C
{
public:
        D(int x): A(x) {cout << "D(int)\n";}
        ~D(void) {cout << "~D()\n";  }
        void Print(void){cout << x << endl;};
};
```

Moştenirea multiplă – studiu de caz
Problema diamantului

- Apel main:

```
A(int)
B()
C()
D(int)
D::x=5
sizeof(d)=12
~D()
~C()
~B()
~A()
```

O clasă moşteneşte 2 clase care au o funcție membră cu aceaşi semnătură




```
d.Print();
//...
error C2385: ambiguous access of 'Print' in 'D'
```

- Soluții:
  - redefinirea funcției respective în clasa derivată

```
void D::Print(void)
{
        cout << "D::x=" <<x<< endl;
}
```

  - utilizarea operatorului de rezoluție apartenență la domeniu pentru a specifica in mod clar domeniul de care aparține funcția

```
d.B::Print();
```

Ierarhii de clase

- Odată cu moștenirea trebuie rezolvată o problemă: Având un pointer la clasa de bază (*Base**), cărei clase derivate aparține obiectul pointat. Sunt 4 soluții:
  - Asigura-te că pointerul pointează către obiecte de același tip (nu se recomandă dar poate fi o soluție pentru containere omogene)
  - Plasarea unui membru în clasa de bază pentru a caracteriza tipul de date.
  - A se utiliza *dynamic_cast* pentru convertirea pointerului
  - A se utiliza funcții virtuale

```
class Angajat
{
protected:
        enum TipAngajat{man, ang};
        TipAngajat angType;
public:
        Angajat(void): angType(ang){/*…*/};
        //…
};
```

Ierarhii de clase

```
class Manager : public Angajat
{    //…
public:
    Manager(void){angType=man; /*…*/};
    //…
};
void Angajat::Print(void)const{
    switch(angType)
    {
    case ang:
        cout << "Angajat -> Nume: " << firstName << " " <<
                middleInitial << ". " << familyName << " \n";
        break;
    case man:
        cout << "Manager -> Nume: " << firstName << " " <<
                middleInitial << ". " << familyName;
        const Manager*p = static_cast<const Manager*>(this);
        cout << ", level" << (((Manager&))(*p)).GetLevel() <<"\n";
        break;
    }
}
```

Ierarhii de clase

```
int main(void)
{
    Date da(5,10,1977);
    Date dm(10,5,1968);
    cout<<"**************************************\n";

    Manager m("Gica", "Popescu", 'G', dm, 3, 10, 8);
    Angajat a("Mitica", "Gheorghe", 'I', da, 1);

    Angajat *pa;
    pa = &a;
    pa->Print();

    pa = &m;
    pa->Print();

    return 0;
}
```

Ierarhii de clase

Apel main:

```
Angajat -> Nume: Mitica I. Gheorghe
Manager -> Nume: Gica G. Popescu, level 8
```

- Această soluție nu se recomandă deoarece prezintă probleme de întreținere, fiind o soluție generatoare de erori

Funcții virtuale

- Utilizarea funcțiilor virtuale elimină problemele generate de soluția cu membru de caracterizare a tipului obiectului, permițând programatorului să declare funcții în clasa de bază ce pot fi redefinite în fiecare clasă
- Compilatorul și linkeditorul va garanta corespondența corectă între obiecte și funcțiile aferente.

```
class Angajat
{
private:
    //…
public:
    //…
    virtual void Print(void)const;
};
```

Funcții virtuale

```
int main(void)
{
    Date da(5,10,1977);
    Date dm(10,5,1968);

    cout<<"*****************************************\n";
    Manager m("Gica", "Popescu", 'G', dm, 3, 10, 8);
    Angajat a("Mitica", "Gheorghe", 'I', da, 1 );

    Angajat *pa;
    pa = &a;
    cout << "Angajat -> ";
    pa->Print();

    pa = &m;
    std::cout << "Manager -> ";
    pa->Print();

    return 0;
}
```

Funcții virtuale

- Cuvântul cheie virtual indică faptul că *`Print()`* se comportă ca o interfață atât pentru funcția *`Print()`* definită în clasa *`Angajat`* cât și pentru funcția *`Print()`* definită în clasa *`Manager`* derivată din *`Angajat`*.
- Compilatorul va asigura că pentru un obiect *`Angajat`* va fi apelată funcția *`Print()`* corespunzătoare, dacă în clasele derivate aceasta a fost definită.
- Toate funcțiile *`Print()`* redefinite trebuie să aibă aceeași semnătură
- O funcție virtuală trebuie definită în clasa unde a fost declarată (cu excepția cazurilor când sunt declarate ca fiind virtuale pure)
- Determinarea funcției *`Print()`* corespunzătoare, independent de tipul *`Angajat`*-ului se numește **polimorfism**.

Polimorfism

- Un tip de date cu funcții virtuale se numește *tip de date polimorfic* sau mai exact ***run-time polimorphic type***.
- Pentru a obține un comportament polimorfic obiectele trebuie manipulate prin intermediul pointerilor sau referințelor
- O funcție ce suprascrie o funcție virtuală devine la rândul ei virtuală.
- Pentru a implementa polimorfismul, compilatorul trebuie să rețină unele informații în fiecare obiect de tip *`Angajat`* și să le folosească pentru a apela funcția *`Print()`* corespunzătoare.
- Conectarea unui apel al unei funcții de corpul acesteia se numește legătura

Polimorfism

- Când legătura se realizează înainte de rularea unui program(de către compilator și linkeditor), se numește **legare timpurie sau legare statică** (*early binding*)
- Legarea implicită în C++ este cea statică
- Polimorfismul este un mecanism prin care legarea dintre apelul unei metode și corpul acesteia se face la momentul rulării (**legarea dinamică**) (*late binding*)
- În mod uzual compilatorul convertește numele unei funcții virtuale într-un index din cadrul unui tabel de pointeri la funcții. Acel tabel este numit *tabelul cu funcții virtuale* sau mai simplu *vftbl*.
- Fiecare clasă cu funcții virtuale are propriul său *vftbl*

Tabela funcțiilor virtuale

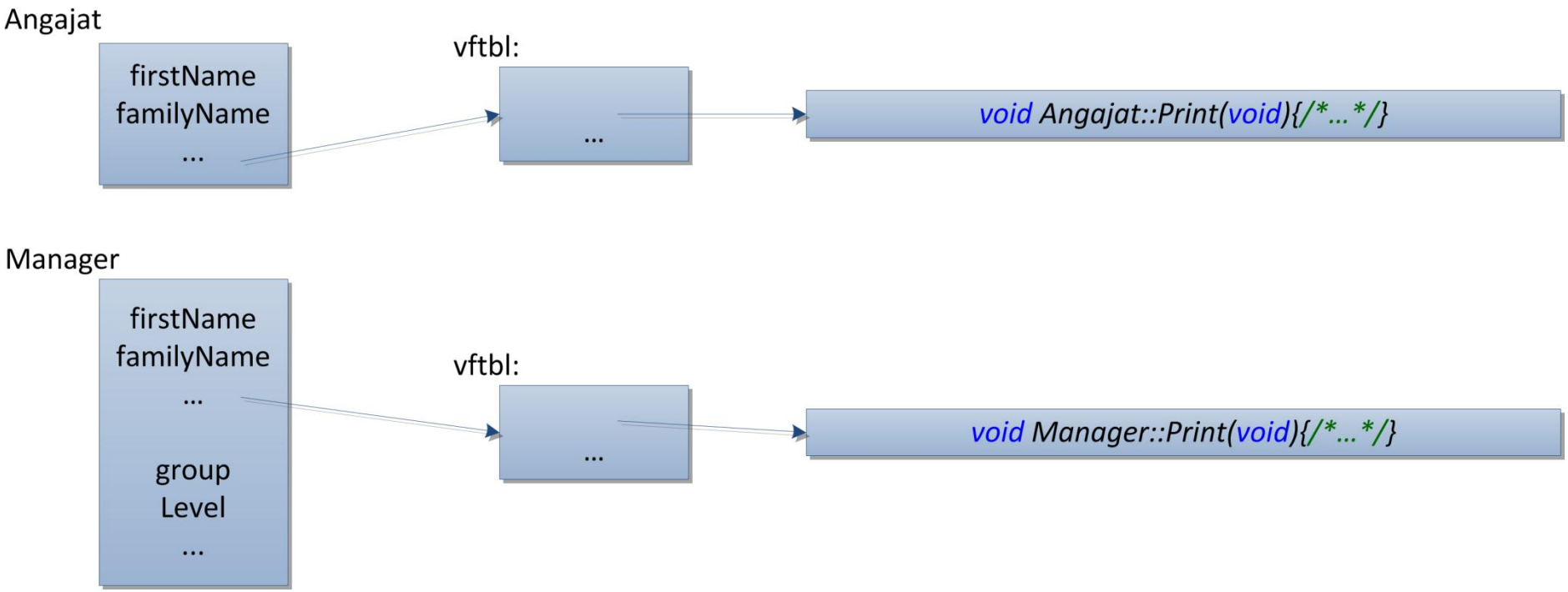


Tabela funcțiilor virtuale

- Conține pointeri către funcțiile virtuale
- Este creată pentru fiecare clasă ce conține funcții membre virtuale sau suprascrie funcții virtuale
- Există numai una pentru o anumită clasă
- Există clase pentru care nu este creată
- Când un obiect este creat, se adaugă un membru ascuns, un pointer către această tabelă virtuală
- Compilatorul generează automat codul în constructori pentru inițializarea pointerului către această tabelă
- Se accesează în momentul când se execută o funcție virtuală

Vă mulțumesc !