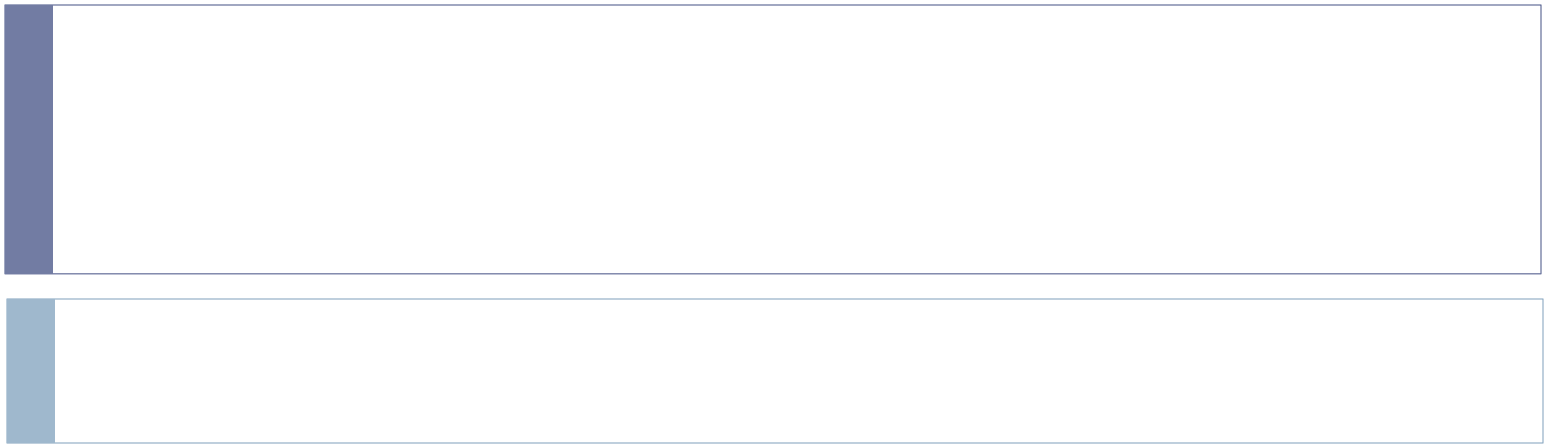
PROGRAMARE ORIENTATĂ OBIECT

Curs 6

Moştenire



Introducere

- Limbajul C++ a împrumutat ideea claselor și ierarhiei de clase din limbajul Simula. De asemenea a împrumutat ideea de modelare a noilor concepte/tipuri de date prin intermediul claselor.
- Un concept (noțiune, idee) nu există de sine stătător ci coexistă cu alte concepte și ceea ce este mai important, în relație cu acestea.
- Ex. conceptul de mașină introduce alte noțiuni: roată, motor, șofer, pieton, camion, ambulanță, drum, ulei etc.
- Având în vedere utilizarea claselor pentru reprezentarea conceptelor, problema care apare este cum să fie reprezentată relația dintre concepte.
- Noțiunea de clasă derivată este utilizată pentru a explica similitudinile dintre clase, relația erarhică.

Introducere

- Conceptele de cerc și triunghi au ceva în comun și anume conceptul de figură. Astfel se definește clasa `Cerc` și clasa `Triunghi` ca având clasa `Figura` în comun.
- Clasa comună `Figura` este referită clasa de bază sau superclasă și clasa derivată (`Cerc` sau `Triunghi`) din clasa `Figura` ca fiind clasa derivată sau subclasă.
- Limbajul de programare oferă facilități pentru construirea de noi clase pe baza celor existente:
  - *Implementation inheritance*: utilizarea facilităților furnizate de clasa de bază în clasa derivată
  - *Interface inheritance*: utilizarea diferitelor clase derivate prin intermediul interfeței furnizate de clasa de bază comună (run-time polimorfism sau dynamic polimorfism).

Se considera clasa Date

```
const int monthDays[]={31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

int CheckBisect(int year)
{
    return (((year%4)==0)&&(!(((year%100)==0)&&((year%400) != 0))));
}

class Date
{
    int d, m, y;
public:
    Date(int dd=0, int mm=0, int yy=0):d(dd), m(mm), y(yy) {};
    //metode Get Set
    Date operator-(const Date&dc){ /*…*/};
};
```

Clase derivate

- Se consideră un program ce gestionează persoanele angajate dintr-o firmă:

```
class Angajat
{
    char *firstName;
    char *familyName;
    char middleInitial;
    Date hiringDate;
    short int departament;
};
```

```
class Manager
{
    Angajat ang;//datele de angajat
                //ale managerului
    Angajat *group;//oameni in
                   //subordine
    short int level;
};
```

- Un manager este un *Angajat*. Datele de *Angajat* sunt stocate în membrul *ang* al obiectul *Manager*.
- Nu este nimic care să „spună” compilatorului că *Manager* este un *Angajat*.
- De asemenea un *Manager** nu este un *Angajat**. Nu se pot pune *Manager** într-un vector cu *Angajat**

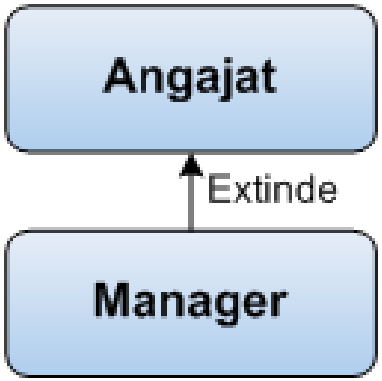
Clase derivate

- Se poate converti explicit *Manager** în *Angajat** sau pur și simplu să fie pusă adresa membrului *ang* într-o listă de angajați, dar nu se recomandă
- Abordarea corectă este de a afirma în mod explicit că *Manager* este un *Angajat* dar cu noi informații adaugate.

```
class Manager : public Angajat
{
        Angajat *group;        //oameni in subordine
        short int level;
};
```

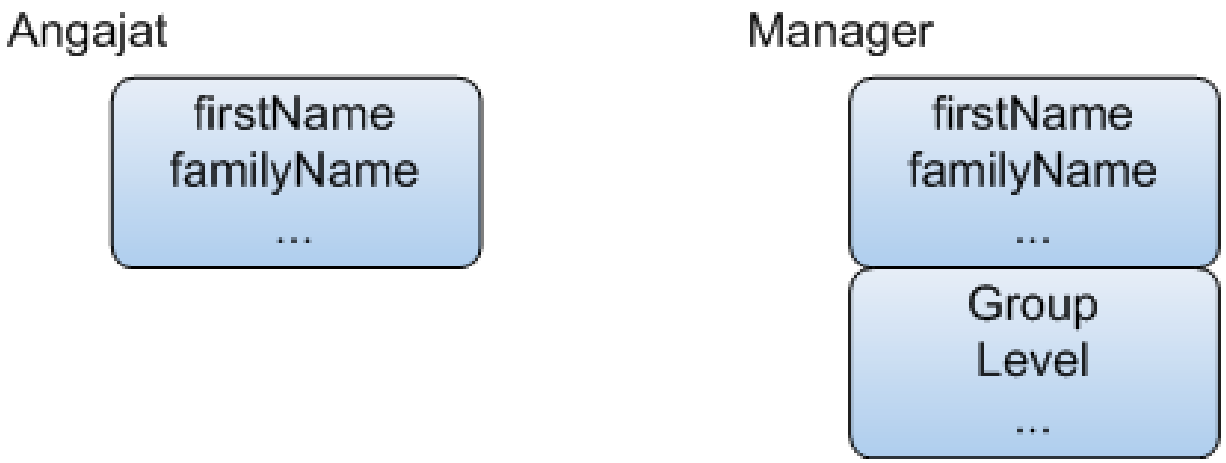
Clase derivate

- Clasa *Manager* are membri *firstName*, *departament* etc. și în *plus group*, *level* etc.
- Derivarea este reprezentată grafic printr-o săgeată de la clasa derivată către clasa de bază



Clase derivate

- În mod obișnuit se spune că o clasă derivată moștenește proprietățile din clasa de bază, motiv pentru care relația se numește moștenire.
- O implementare populară și eficientă a noțiunii de clasă derivată este un obiect al clasei derivate reprezentat ca un obiect al clasei de bază cu informații în plus ce aparțin doar clasei derivate, adăugate la final.



Clase derivate

- Derivând *Manager* din *Angajat* face ca *Manager* să fie un subtip al tipului *Angajat*. Astfel se poate crea un vector de angajați în care unii dintre ei sunt manageri

```
Angajat a1;
Manager m1;
Angajat v[2] = {a1, m1};
```

Clase derivate

- Având în vedere că *Manager* este un *Angajat* atunci și *Manager** poate fi folosit ca un *Angajat**. Similar și *Manager&* poate fi folosit ca un *Angajat&*.
- Totuși un *Angajat* nu este necesar un *Manager* astfel un *Angajat** nu poate fi folosit ca un *Manager**.
- În general dacă o clasă *Derivată* are o clasă publică *Bază*, atunci un pointer *Derivată** poate fi atribuit unui pointer de tip *Bază** fără conversie explicită de tip. Invers, din *Bază** în *Derivată** conversia trebuie realizată în mod explicit.
- O clasa trebuie definită înainte de a fi folosită ca o clasă de bază

Moştenirea C++

- Pentru a deriva o clasă dintr-alta, se folosește operatorul ":" la declararea clasei derivate după următorul format:

```
class ClasaDerivata : public ClasaBaza
{ … }
```

- Specificatorul de acces (*public*) poate fi înlocuit de unul dintre ceilalți doi specificatori *private* sau *protected*

```
class ClasaBaza
{
        //membri clasa de baza
};

class ClasaDerivata: [public|protected|private] ClasaBaza
{
        //membrii clasa derivata
};
```

Moştenirea – specificatori de acces

Modificatorii de protecție utilizați în lista claselor de bază, definesc protecția în clasa derivată a elementelor moștenite. O clasă derivată va moșteni membrii clasei de bază, dar accesul la aceștia va fi restricționat de modificatorii de protecție.

Accesul în clasa de bază	**Specificatorul de acces din lista claselor de bază**	**Accesul în clasa derivată a membrului moştenit**
Private	Private	inaccesibil
Protected		private
Public		private
Private	Protected	inaccesibil
Protected		protected
Public		protected
Private	Public	inaccesibil
Protected		protected
Public		public

Moştenirea – specificatori de acces

- La moștenirea private, membrii moșteniti cu protecția *protected* sau *public*, vor avea protecția *private* în clasa derivată. Această înseamnă că nu vor mai putea fi accesați de o clasă derivată din clasă ce derivă din clasa de bază de unde sunt moșteniți. Practic această tehnică închide ierarhia, și posibilitatea de extindere a acesteia.
- **De obicei, o ierarhie de clase nu este o ierarhie finală, ea putând fi dezvoltată adăugând clase noi, care derivă din clasele terminale.**
- Pentru a facilita extinderea ierarhiei fără probleme se folosește moștenirea publică.

Funcții membre

```
class Angajat
{
private:
    char *firstName;
    char *familyName;
    char middleInitial;
    Date hiringDate;
    short int departament;
public:
    void Print(void)const;
    char* GetFullName(void);
};

char* Angajat::GetFullName(void)
{
    char *strTmp = new char[strlen(firstName)+strlen(familyName) + 4];
    sprintf(strTmp,"%s %c. %s", firstName, middleInitial, familyName);
    return strTmp;
}
```

```
class Manager : public Angajat
{
private:
    Angajat *group;
    short int level;
public:
    void Print(void)const;
};
```

Funcții membre

- O metodă a clasei derivate poate folosii membri publici si protected din clasa de bază ca și cum ar apartine clasei derivate

```
void Manager::Print(void)
{
    char *fullName = GetFullName();
    cout << "Numele: " << fullName <<"\n";
    delete []fullName;
}
```

- Totuși o clasă derivată nu poate accesa membri privați ai clasei de bază (eroare de compilare)

```
void Manager::Print(void)
{
    cout << "Numele: " << familyName <<"\n";
}
```

- Acolo unde este necesar, membri privați din clasa de bază pot fi declarați ca fiind *protected* astfel încât să fie accesibili și în clasele derivate.

Funcții membre

- Cea mai bună soluție de implementare ar fi:

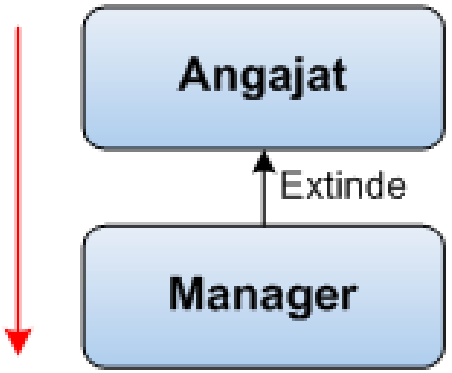
```
void Manager::Print(void)
{
        Angajat::Print(); //afiseaza informatii Angajat
        cout << level <<"\n"; //afiseaza informatii specifice Manager
        //...
}
```

- Având în vedere că *Print* a fost redefinită în clasa Manager trebuie specificat în mod clar care funcție membră să fie apelată

```
Angajat::Print();
```

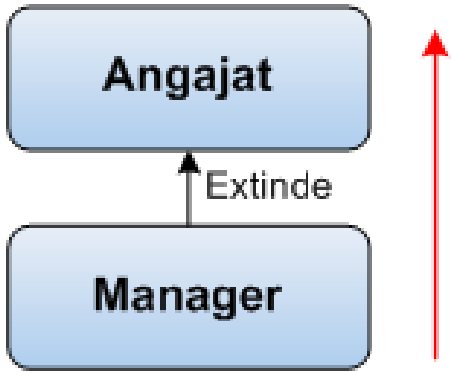
Constructori şi destructori

- Ordinea de apelare a constructorilor pentru un obiect dintr-o clasă derivată: de la clasa de bază către clasa derivată
  - Constructorul clasei de bază
  - Constructorul clasei derivate



Constructori şi destructori

- Ordinea de apelare a destructorilor pentru un obiect dintr-o clasă derivată: de la clasa derivată către clasa de bază
  - Destructorul clasei derivate
  - Destructorul clasei de bază



Constructorii apelați implicit

- Pentru o instanță a unei clase de bază, constructorul apelat implicit este constructorul fără listă de argumente (dacă există).
- Prin apeluri de forma:

```
Manager::Manager( … ) : Angajat( … ) {}
```

se poate controla ce constructor din clasa de bază va fi folosit (se forțează compilatorul pentru a apela un anumit constructor)

Moştenirea “în lanț” (în cascadă)

```
class A
{
public:
    A()
    { cout << "A" << endl; }
};

class B: public A
{
public:
    B()
    { cout << "B" << endl; }
};

class C: public B
{
public:
    C()
       { cout << "C" << endl; }
};
```

```
int main()
{
     C objC;
}
```

La rulare, se va afișa:

A

B

C

Suprascrierea funcțiilor din clasa de bază

```
void Angajat::Print(void)const
{
    std::cout<<"Nume: "<<firstName<<" "<<middleInitial<<". „
                                        <<familyName<<" \n";
}
void Manager::Print(void)const
{
    std::cout<<"Nume: "<<firstName<<" "<<middleInitial<<". „
                                    <<familyName<<" \n";
    std::cout << level <<"\n";
       //...
}
```

sau

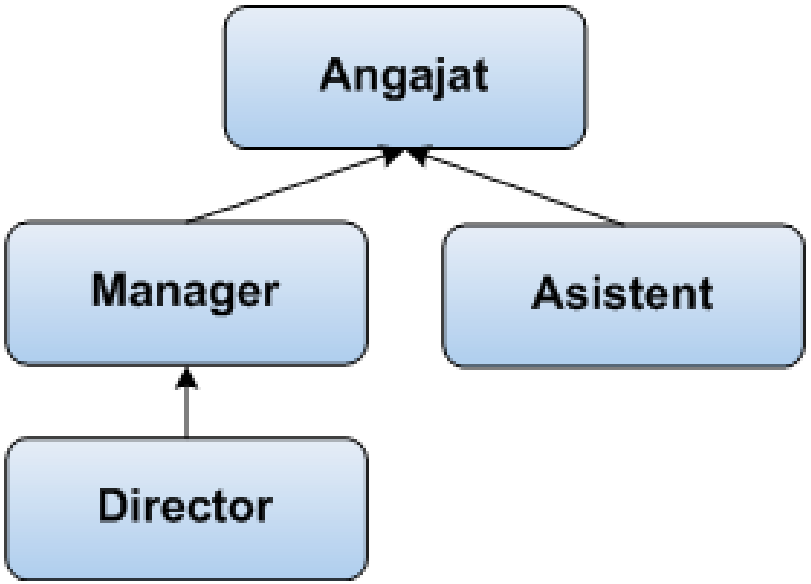
```
void Manager::Print(void)const
{
       Angajat::Print();
       std::cout << level <<"\n";
       //...
}
```

Ierarhie de clase

- O clasă derivată poate fi o clasă de bază:

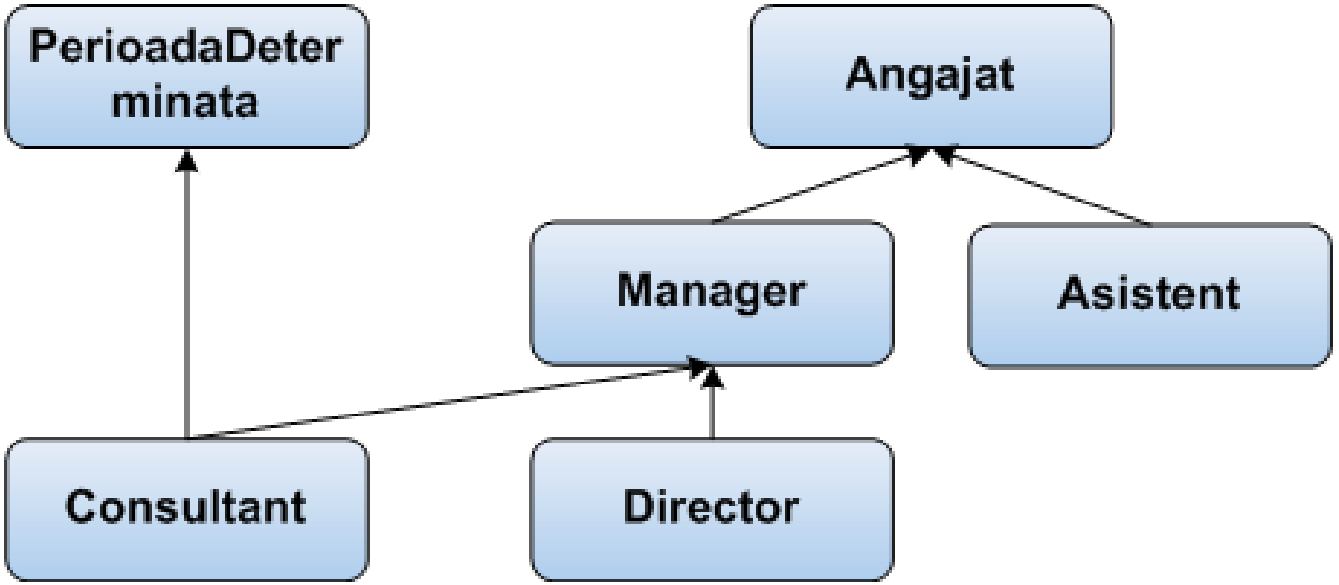
```
class Angajat { /* ... */ };
class Manager : public Angajat { /* ... */ };
class Director : public Manager { /* ... */ };
class Asistent : public Angajat { /* ... */ };
```

- Un de set de clase relaționate în acest fel se numește ierarhie de clase



Ierarhie de clase

- În general o ierarhie de clase are forma unui arbore dar poate fi și sub forma unui graf



- Moștenirea multiplă apare atunci când o clasă derivată are mai multe clase de bază.

```
class PerioadaDeterminata { /* ... */ };
class Consultant : public PerioadaDeterminata, pu
Manager
{ /* ... */ };
```

Sintaxă

```
class Bi
{
        //membri clasa de baza
};

class D: [public|protected|private]B1,…,[public|protected|private]Bn
{
        //membrii clasa derivata
};
```

Moştenire multiplă

```
class B1
{
protected:
    int m_b1;
public:
    B1(int b1) : m_b1(b1) { cout << "B1(int)\n";}
    ~B1(void) {cout << "~B1()\n";}
    void Print(void) {cout << m_b1 << endl;}
};
class B2
{
protected:
    float m_b2;
public:
    B2(float b2): m_b2(b2) { cout << "B2(float)\n";}
    ~B2(void) {cout << "~B2()\n";}
    void Print(void) {cout << m_b2 << endl;}
};
```

Moştenire multiplă

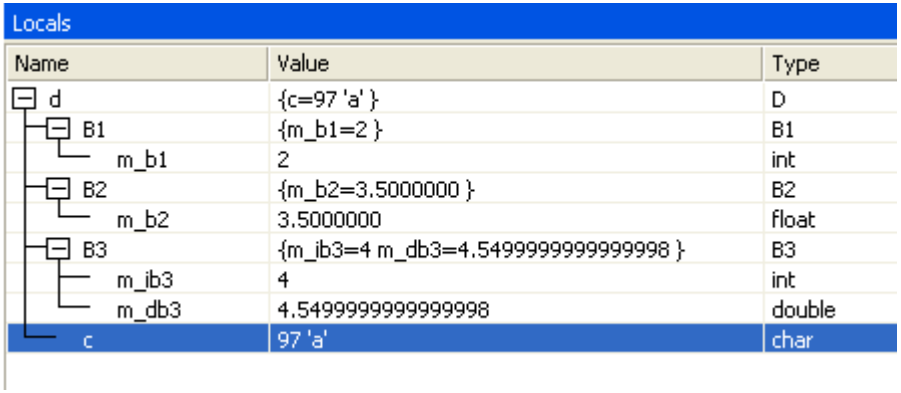
```
class B3{
protected:
    int m_ib3;
    double m_db3;
public:
    B3(int  ib3,  double  db3):  m_ib3(ib3),  m_db3(db3){cout<<
                                        "B3(int, double)\n";}
    ~B3(void) {cout << "~B3()\n";}
    void Print(void) {cout<<m_ib3<<endl; cout<<m_db3<<endl;}
};
```

Moştenire multiplă

```
class D: public B1, private B2, protected B3 {
protected:
        char c;
public:
        D(int i1, float f1, int i2, double d1, char cc):B3(i2,
                        d1), B2(f1), B1(i1),
                        c(cc){cout<<"D(int,float,int,double)\n";}
        ~D(void) { cout << "~D()\n";}
        void afisare() {
                B1::Print();
                B2::Print();
                B3::Print();
                cout << c << endl;
        }
};
```

Moştenire multiplă

```
int main(void)
{
        D d(2, 3.5f, 4, 4.55, 'a');
        d.Print();
        return 0;
}
```



Locals

Name	Value	Type
d	{c=97 'a' }	D
B1	{m_b1=2 }	B1
m_b1	2	int
B2	{m_b2=3.5000000 }	B2
m_b2	3.5000000	float
B3	{m_ib3=4 m_db3=4.5499999999999998 }	B3
m_ib3	4	int
m_db3	4.5499999999999998	double
c	97 'a'	char

Rulare:

Apel constructori:

B1(int)
B2(float)
B3(int, double)
D(int, float, int, double)

2
3.5
4
4.55
a

Apel destructori:

~D()
~B3()
~B2()
~B1()

Vă mulțumesc !