Curs 5 Moștenire. Polimorfism

- **Moștenire in C++**
- **Diagrame UML de clase**
- **Polimorfism**

Moștenire

Moștenirea permite definirea de clase noi (clase derivate) reutilizând clase existente (clasă de bază). Clasa nou creată moșteneste comportamentul (metode) și caracteristicile (variabile membre, starea) de la clasa de bază

Dacă A și B sunt două clase unde B moșteneste de la clasa A (B este derivat din clasa A sau clasa B este o specializare a clasei A) atunci:

- clasa B are toate metodele si variabilele membre din clasa A
- clasa B poate redefini metode din clasa A
- clasa B poate adauga noi membrii (variabile, metode) pe lângă cele moștenite de la clasa A.

```
class Person {
public:
	Person(string cnp, string name);
	const string& getName() const {
		return name;
	}

	const string& getCNP() const {
		return cnp;
	}
	string toString();
protected:
	string name;
	string cnp;

};
```

```
class Student: public Person {
public:
	Student(string cnp, string name,
string faculty);
	const string& getFaculty() const {
		return faculty;
	}
	string toString();
private:
	string faculty;
};
```

Moștenire simplă. Clase derivate.

Dacă clasa B moștenește de la clasa A atunci:

- orice obiect de tip B are toate variabilele mebre din clasa A
- funcțiile din clasa A pot fi aplicate si asupra obiectelor de tip B (daca vizibilitatea permite)
- clasa B poate adăuga variabile membre și sau metode pe lângă cele moștenite din A

```
class A:public B{
….
}
```

clasa B = Clasă de bază (superclass, base class, parent class)

clasa A = Clasă derivată (subclass, derived class, descendent class)

membrii (metode, variabile) moșteniți = membrii definiți în clasa A și nemodificați în clasa B

membrii redefiniți (overridden) = definit în A și în B (în B se crează o nouă definiție)

membrii adăugați = definiți doar în B

Vizibilitatea membrilor moșteniți

Dacă clasa A este derivat din clasa B:

- clasa A are acces la membri publici din B
- clasa A nu are acces la membrii privați din B

```
class A:public B{
….
}
```

public membrii publici din clasa B sunt publice și in clasa B

```
class A:private B{
…
}
```

private membrii publici din clasa B sunt private în clasa A

```
class A:protected B{
…
}
```

protected membrii publici din clasa B sunt protejate în clasa A (se vad doar in clasa A și în clase derivate din A).

Modificatori de access

Definesc reguli de access la variabile membre și metode dintr-o clasă

`public`: poate fi accesat de oriunde

`private`: poate fi accesat doar în interiorul clasei

`protected`: poate fi accesat în interiorul clasei și în clasele derivate.

`protected` se comportă ca și **`private`**, dar se permite accessul din clase derivate

Access	**`public`**	**`protected`**	**`private`**
clasa	Da	Da	Da
clasa derivată	Da	Da	Nu
În exterior	Da	Nu	Nu

Constructor/Destructor în clase derivate

- Constructorii și destructorii nu sunt moșteniți
- Constructorul din clasa derivată trebuie sa apeleze construcorul din clasa de baza. Sa ne asigurăm ca obiectul este initializat corect.
- Similar și pentru destructor. Trebuie sa ne asiguram ca resursele gestionate de clasa de bază sunt eliberate.

```
Student::Student(string cnp, string name, string faculty) :
              Person(cnp, name) {
       this->faculty = faculty;
}
```

- Dacă nu apelăm explicit construcorul din clasa de bază, se apelează automat constructorul implicit
- Dacă nu exista constructor implicit se genereaza o eroare la compilare

```
Student::Student(string cnp, string name, string faculty) {
       this->faculty = faculty;
}
```

Se apelează destructorul clasei de bază

```
Student::~Student() {
       cout << "destroy student\n";
}
```

Initializare.

Cand definim constructorul putem initializa variabilele membre chiar înainte sa se execute corpul constructorului.

```
Person::Person(string c, string n) :
            cnp(c), name(n) {
}
```

Initializare clasă de bază

```
Manager(std::string name, int yearInFirm, float payPerHour, float bonus) :
                  Employee(name, yearInFirm, payPerHour) {
      this->bonus = bonus;
}
```

Apel metodă din clasa de bază

```
float Manager::payment(int hoursWorked) {
      float rez = Employee::payment(hoursWorked);
      rez = rez + rez * bonus;
      return rez;
}
```

Creare /distrugere de obiecte (clase derivate)

Creare

- se alocă memorie suficientă pentru variabilele memre din clasa de bază
- se alocă memorie pentru variabile membre noi din clasa derivată
- se apelează constructorul clasei de bază pentru a inițializa atributele din clasa de bază
- se execută constructorul din clasa derivată

Distrugere

- se apelează destructorul din clasa derivată
- se apelează destructorul din clasa de bază

Principiul substituției.

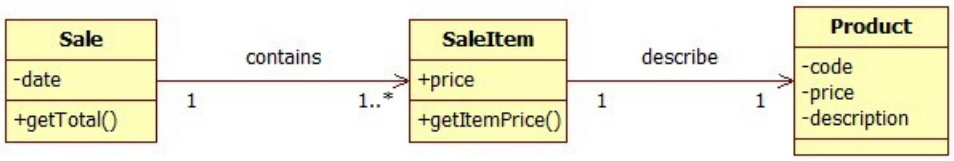
Un obiect de tipul clasei derivate se poate folosi în orice loc (context) unde se cere un obiect de tipul clasei de bază. (upcast implicit!)

```
Person p = Person("1", "Ion");
cout << p.toString() << "\n";

Student s("2", "Ion2", "Info");
cout << s.toString() << "\n";

Teacher t("3", "Ion3", "Assist");
cout << t.getName() << " " << t.getPosition() << "\n";

p = s;
cout << p.getName() << "\n";

p = t;
cout << p.getName() << "\n";

s = p;//not valid, compiler error
```

Pointer

```
Person *p1 = new Person("1", "Ion");
cout << p1->getName() << "\n";

Person *p2 = new Student("2", "Ion2", "Mat");
cout << p2->getName() << "\n";

Teacher *t1 = new Teacher("3", "Ion3", "Lect");
cout << t1->getName() << "\n";

p1 = t1;
cout << p1->getName() << "\n";

t1 = p1;//not valid, compiler error
```

Diagrame UML (Is a vs Has a)



- Un Sale are una sau mai multe SaleItem
- Un SaleItem are un Product

Relaţia de asociere UML (Associations): Descriu o raleţie de dependenţă structurală între clase

Elemente posibile:

- nume
- multiplicitate
- nume rol
- uni sau bidirecţional

Tipuri de relaţii de asociere

- Asociere
- Agregare (compoziţie) (whole-part relation)
- Dependenţa
- Moştenire

Are (has a):

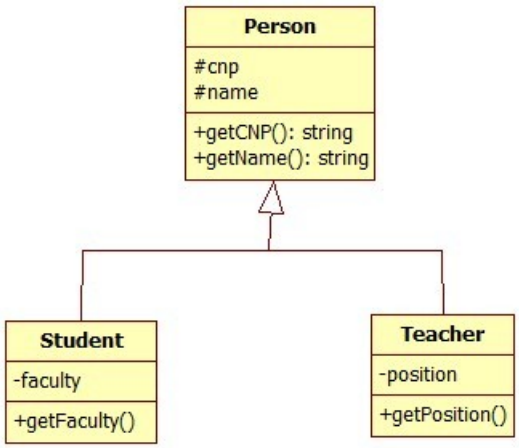
- Orice obiect de tip A are un obiect B.
- SaleItem are un Product. Persoana are nume (string)
- in cod apare ca şi o variabilă membră

Este ca şi (is a ,is like a):

- Orice instanţa de tip A este şi de tip B
- Orice student este o persoană
- se implementează folosind moştenirea

Relaţia de specializare/generalizare – Reprezentarea UML .

Folosind moștenirea putem defini hierarhii de clase



Studentul este o Persoană cu câteva atribute adiționale
Studentul moștenește (variabile și metode) de la Persoană
Student este derivat din Persoană. Persoana este clasă de bază,
Student este clasa derivată

Persoana este o generalizare a Studentului
Student este o specializare a Persoanei

Suprascriere (redefinire) de metode.

Clasa derivată poate redefini metode din clasa de bază

`string Person::toString() {`
`  return "Person:" + cnp + " "`
`+ name;`
`}`
	`string Student::toString() {`
	`  return "Student:" + cnp + " " +`
	`name + " " + faculty;`
	`}`

`Person p = Person("1", "Ion");` `cout << p.toString() << "\n";`	`Student s("2", "Ion2", "Info");` `cout << s.toString() << "\n";`

În clasa derivata descriem ce este specific clasei derivate, ce diferă față de clasa de bază

Suprascriere (overwrite) ≠ Supraîncărcare (overload)

```
string Person::toString() {
      return "Person:" + cnp + " " + name;
}

string Person::toString(string prefix) {
      return prefix + cnp + " " + name;
}
```

toString este o metodă supraîncărcată(**`toString()`**,**`toString`**`(string prefix)`)

Polimorfism

Proprietatea unor entităţi de:

- a se comporta diferit în funcţie de tipul lor
- a reacţiona diferit la acelaşi mesaj

Obiecte din diverse clase care sunt legate prin relaţii de moştenire să răspundă diferit la acelaşi mesaj (apel de metodă).

Proprietate a unui limbaj OO de a permite manipularea unor obiecte diferite prin intermediul unei interfeţe comune

Tipul declarat vs tipul actual

Orice variabilă are un tip declarat (la declararea variabilei se specifică tipul).
În timpul execuției valoarea referită de variabila are un tip actual care poate diferi de tipul declarat

```
Student s("2", "Ion2", "Info");
Teacher t("3", "Ion3", "Assist");
Person p = Person("1", "Ion");

cout << p.toString() << "\n";

p = s;
cout << p.toString() << "\n";

p = t;
cout << p.toString() << "\n";
```

Tipul declarat pentru p este Persoană, dar în timpul execuției p are valori de tip Person, Student și Teacher.

```
string Person::toString() {
	return "Person:" + cnp + " " + name;
}
string Student::toString() {
	return "Student:" + cnp + " " + name + " " + faculty;
}
string Teacher::toString() {
	string rez = Person::toString();
	return "Teacher " + rez;
}
Student s("2", "Ion2", "Info");
Person* aux = &s;
cout << aux->toString() << "\n";
Person p = Person("1", "Ion");
aux = &p;
cout << aux->toString() << "\n";
```

- Person,Student,Teacher are metoda toString , fiecare clasă definește propria versiune de toString.

- Sistemul trebuie sa determine dinamic care dintre variante trebuie executată în momentul în care metoda toString este apelată.

- Decizia trebuie luată pe baza tipului actual al obiectului.

- Funcţionalitate importantă (prezent în limbaje OO) numit legare dinamică - dynamic binding (late binding, runtime binding).

Legare dinamică (Dynamic binding).

Legarea (identificarea) codului de executat pe baza numelui de metode se poate face:

- în timpul compilării => legare statică (static binding)
- în timpul execușiei => legare dinamică (dynamic binding)

Legare dinamică:

- selectarea metodei de executat se face timpul execuției.
- Când se apelează o metodă, codul efectiv executat (corpul funcției) se alege la momentul execuției (la legare statică decizia se ia la compilare)
- legarea dinamica în C++ funcționeaza doar pentru referințe și pointeri
- În C++ doar metodele virtuale folosesc legarea dinamică

Metode virtuale.

Legarea dinamică în c++: Folsind metode virtuale
O metodă este declarată virtual în casa de bază:
virtual <function-signature>

- metoda suprascrisă în clasele derivate are legarea dinamică activată
- metoda apelată se va decide în funcșie de tipul actual al obiectului (nu în funcșie de tipul declarat).

- Constructorul nu poate fi virtual – pentru a crea un obiect trebuie sa ştim tipul exact
- Destructorul poate fi virtual (este chiar recomandat sa fie când avem hierarhii de clase)

```
class Person {
protected:
	string name;
	string cnp;

public:
	Person(string cnp, string name);
	virtual ~Person();

	const string& getName() const {
		return name;
	}

	const string& getCNP() const {
		return cnp;
	}
	virtual string toString();
	string toString(string prefix);
};
```

Mecanism C++ pentru polimorfism

Orice obiect are atașat informații legate de metodele obiectului

Pe baza acestor informații apelul de metodă este efectuat folosind implementarea corectă (cel din tipul actual). Orice obiect are referință la un tabel prin care pentru metodele virtuale se selectează implementarea corectă.

Orice clasă care are cel puțin o metodă virtuală (clasă polimorfică) are un tabel numit VTABLE (virtual table). VTABLE conține adresse la metode virtuale ale clasei.

Când invocăm o metodă folosind un pointer sau o referință compilatorul generează un mic cod adițional care în timpul excuției o sa folosească informația din VTABLE pentru a selecta metoda de executat.

Destructor virtual

- Destructorul este responsabil cu dealocarea resurselor folosite de un obiect
- Dacă avem o hierarhie de clasă atunci este de dorit să avem un comportament polimorfic pentru destructor (să se apeleze destructorul conform tipului actual)
- Trebuie să declarăm destructorul ca fiind virtual

Moștenire multiplă

În C++ este posibil ca o clasă să aibă multiple clase de bază, să moștenească de la mai multe clase

```
class Car : public Vehicle , public InsuredItem {

};
```

Clasa moștenește din toate clasele de bază toate atributele.
Moștenirea multipla poate fi periculoasă și în general ar trebui evitat

- se poate moșteni același atribut de la diferite clase
- putem avea clase de bază care au o clasă de bază comună

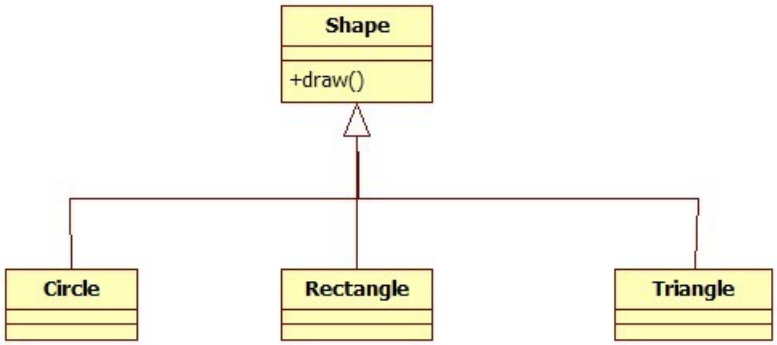
Funcții pur virtuale

Funcțiile pur virtuale nu sunt definite (avem doar declarația metodei). Folosim metode pur virtuale pentru a ne asigura că toate clasele derivate (concrete) o sa definească metoda.

```
class Shape {
public:
     Shape();
     virtual ~Shape();
     virtual void draw() = 0;//pure virtual
};
```

=0 indică faptul ca nu există implementare pentru această metodă în clasă.
Clasele care au metode pur virtuale nu se pot instanția

Shape este o clasă abstractă - definește doar interfața, dar nu conține implementări.



Clase abstracte

O clasă abstractă poate fi folosită ca și clasă de bază pentru o colecție de clase derivate;
Oferă:

- o interfață comună pentru clasele derivate (metodele pur virtuale se vor implementa în clasele derivate)
- pot conține atribute comune tuturor claselor derivate

o clasă abstractă nu are instațe
o clasă abstractă are cel puțin o metodă pur virtuală:
`virtual` <return-type> <name> (<parameters>) = 0;

clasă pur abstractă = clasă care are doar metode pur vortuale
clasă pur abstractă = interfață

În UML font italic

Clase care extind clase abstracte

- O clasă derivată dintr-o clasă abstractă mosteneste interfața publică a clasei abstracte
- clasa suprascrie metodele definite în clasa abstractă, oferă implementări specifice pentru funcțiile definite în clasa abstracta
- putem avea instanțe