Laboratorul 5

Moștenire simplă, metode suprascrise (override), interfețe utile

Voi prezenta atât modul în care implementezi de la zero, cât și cheile rapid pentru CLion care ne ajută sa implementăm și navigăm rapid între funcțiile din clasa părinte și funcțiile din clasele moștenitoare (denumite și derivate, dar din fericire nu ca la mate :).)

# Exemple de bază

## Ce este moștenirea?

* Înseamnă să punem într-o clasă specială **atributele și metodele *comune*** mai multor clase.

### ***Exemplu (implementat parțial)***

Construim o bază de date/ aplicație în care vom reține diferite persoane care primesc calificative (elevi, studenți, cursanți privați).

Cu totii au in comun: un vector de note (fiecare nota contine o materie si o valoare intre 1 si 10, care poate sa aibă si virgulă) **si** o metoda care calculeaza media notelor. În schimb, elevii au și date despre ciclul curent în care se afla (primar/gimnazial/liceal), studenții introduc și numele facultății, iar cursanții privați numele cursului la care participă.

***Mai sintetizat:*** fiecare tip de student are un vector de note și o metoda *double* mean(); , dar:

* Un elev introduce și ciclul curent (primar/gimnazial/liceal):
* Un student introduce și numele facultății;
* Un cursant privat introduce și numele cursului la care participă.

*// am pus struct ca sa simplificam putin primul exemplu:*

*struct* Nota {

*int* value;

string subject;

};

*// aceasta este clasa de baza/parinte*

*class* PersoanaEvaluata {

*protected*:

vector<Nota> note;*// atributele din clasa de bază se pun sub eticheta protected:*

*public*:

*// constructor cu toti parametrii*

PersoanaEvaluata(*const* vector<Nota> &note) : note(note) {}

*// metoda mean va fi comuna tuturor claselor derivate din PersoanaEvaluata*

*double* mean() {

*double* sum = 0;

*for*(*auto* nota : note) {

sum += nota.value;

}

*return* sum / note.size();

}

};

Clasa elev, **derivata din PersoanaEvaluata**

*// acestea sunt clasele derivate:*

*class* Pupil : *public* PersoanaEvaluata { *// aceasta este sintaxa pentru „moștenire publică”*

*private*:

*// adaugam noi atribute*

string cycle;

*public*:

*// constructori:*

*// TODO va rog sa-i implementati si in clasele derivate, pentru antrenament:*

Pupil(*const* vector<Nota> &note, *const* string &cycle) : PersoanaEvaluata(note), cycle(cycle) {}

*// adaugam SI noi metode*

*void* showGradeSheet() {

*// TODO va afisa toate notele, sub forma*

*// materie: nota*

*// e.g.*

*// subject: literature, value: 10*

*// subject: mathematics, value: 9*

}

*void* gradesGroupedBySubject() {

*// TODO similar cu showGradeSheet, dar daca aveti doua note la aceiasi materie*

*// hint: step-by-step*

*// puteti prima data crea un vector cu toate materiile (vector<string>)*

*// iar dupa aceea parcurgeti cu un prim for subiectele, iar cu un al doilea for parcurgeti toate notele si le afisati pe cele cu acea materie*

*// e.g. afisare*

*// literature: 10 9*

*// mathematics: 8 7 10*

}

};

1. Puteți continua implementarea clasei Pupil (elev), prin a completa cele doua funcții.

Hint: veți observa ca in clasa Pupil aveți acces la atributul protected ***note.***

1. Completați și clasele Student, CourseTaker (constructori cu toti parametrii)

.

## Override - suprascrierea metodelor

Magazinul iMag desfășoară o tombolă în care sunt oferite cadouri de diverse tipuri:

1. Carduri pentru cumpărături;
2. Vacanțe exotice;
3. Produse favorite.

Să folosim această oportunitate pentru a învăța ce înseamnă **suprascrierea metodelor,** dar și cum putem **folosi metoda din clasa de bază în clasa derivată**.

După cum ați aflat din exercițiul trecut, dacă scriem:

*class* Gift {

*protected*:

string name, giverName;

*public*:

Gift(*const* string &name, *const* string &giverName) : name(name), giverName(giverName) {}

*void* offerGift(string receivedBy) {

cout << "The gift named " << name

<< " was received by " << receivedBy

<< " thanks to the courtesy of " << giverName << '\n';

}

};

**// !! Toti constructorii pot fi generate automat in CLion**

*class* GiftCard : *public* Gift {

*private*:

*int* sum;

*public*:

GiftCard(*const* string &name, *const* string &giverName, *int* sum) : Gift(name, giverName), sum(sum) {}

};

*class* GiftHoliday : *public* Gift {

*private*:

string location;

*public*:

GiftHoliday(*const* string &name, *const* string &giverName, *const* string &location) : Gift(name, giverName),

location(location) {}

};

*class* GiftFavouriteProduct : *public* Gift {

*private*:

string category;

*public*:

GiftFavouriteProduct(*const* string &name, *const* string &giverName, *const* string &category) : Gift(name, giverName),

category(category) {}

};

Atributele name, giverNameși metoda *void* offerGift(string receivedBy)vor putea fi folosite de oricare dintre tipurile de produse, adică:

*int* main() {

Gift simpleGift("No Name", "Eleanor Roosevelt");

GiftCard card("Andrei Popescu", "Popescu Miruna", 300);

GiftHoliday holiday("Ibiza platita!", "Leonard Coste", "Ibiza, Spain");

GiftFavouriteProduct phone("Pentru tine", "Costache Leurdean", "Apple Products");

simpleGift.offerGift("Ioan");

card.offerGift("Persida");

holiday.offerGift("Romeo");

phone.offerGift("Caligula");

}

Va putea rula si va genera output-ul:

The gift named "No Name" was received by "Ioan" thanks to the courtesy of "Eleanor Roosevelt"

The gift named "Andrei Popescu" was received by "Persida" thanks to the courtesy of "Popescu Miruna"

The gift named "Ibiza platita!" was received by "Romeo" thanks to the readof "Leonard Coste"

The gift named "Pentru tine" was received by "Caligula" thanks to the courtesy of "Costache Leurdean"

### ***Suprascrierea metodelor (keyword: override)***

#### **Exemplu**

Cum „**specializăm”** mesajele?

Pasul 1. Adăugăm virtual în fața antetului din Gift

*class* Gift {

*// ...*

*virtual void* offerGift(string receivedBy) {

*// ...*

}

};

Pasul 2. Completăm în derivată specializarea

*class* GiftCard : *public* Gift {

*// ...*

*// adăugăm chiar aici o metodă cu același antet ca în clasa părinte + cuvântul cheie override*

*// în CLion,*

*// dăm Ctrl+O ca să deschidem un meniu în care putem alege metoda din părinte,*

*// dupa care vom completa strict corpul noii metode generate.*

*void* offerGift(string receivedBy) *override* {

*// // inlocuim:*

*//* Gift::offerGift(receivedBy); *// apel catre metoda din Gift*

*// cu codul nostru:*

*cout << "Congratulations " << '"' << receivedBy << '"'*

*<< "! You received a gift card of " << sum << "USD from "*

*<< '"' << giverName << '"'*

*<< '\n';*

}

};

#### 

#### Exercițiu

Implementați și în celelalte clase un mesaj care să se potrivească mai mult cu **fiecare tip de cadou (GiftHoliday, GiftFavouriteProduct)** și care să folosească inclusiv **atributele** specifice clasei derivate

## Aplicații utile -> ideea de interfață

Nu-i așa ca a devenit foarte repetitiv să scriem antetele pentru operator<< si operator>>? Putem sa simplificăm treaba in CLion si elimina erori de scriere indiferent de editor.

**Interfața = clasă în care listăm niște funcții care reprezintă „funcționalitățile unui obiect de acest tip” E.g. o interfață pentru un Magazin poate avea metode precum sell(Product), buy(Product), stock() care trebuie sa existe în orice aplicație/clasă de tip magazin.**

Spre exemplu, IoBase va reprezenta interfața unui obiect care poate fi citit și scris de la tastatură.

*Observație!* Doar metodele membre pot fi suprascrise! Așadar vom folosi metode membre care seamănă cu felul în care am implementat **operator>>**

Pasul 1 implementăm o clasa IoBase in care definim metodele pentru citire si scriere:

*class* IoBase {

*private*:

*public*:

*// metoda citire*

*virtual* istream& read(istream& is) { *// nu este nevoie de obiectul citit, pt ca avem this*

*return* is; *// vom vedea ca in interfete deseori nu avem functionalitati*

}

*// metoda scriere*

*virtual* ostream& write(ostream& os) *const* { *// nu este nevoie de obiectul citit, pt ca avem this*

*return* os; *// vom vedea ca in interfete deseori nu avem functionalitati*

}

};

Pasul 2 (de aplicat in fiecare proiect) moștenim prima clasă in fiecare dintre clasele noastre:

*class* PongGame : *public* IoBase {

*private*:

string player1, player2

*// TODO apasam Ctrl+O sau Generate -> Override Functions...*

*// dupa care selectam toate functiile posibile*

*// (sunt listate doar cele virtuale din părinte)*

*public*:

istream &read(istream &is) *override* {

*return* IoBase::read(is);

}

ostream &write(ostream &os) *const override* {

*return* IoBase::write(os);

}

};

Pasul 3. Păstrăm return IoBase::read(is) sau IoBase::write(os), dar inainte de ele vom scrie codul pentru citire:

*class* PongGame : *public* IoBase {

*// ...*

istream &read(istream &is) *override* {

IoBase::read(is); **// in general, punem la inceputul apelul**

cout << "player1: ";

is >> player1;

cout << "player2: ";

is >> player2;

*return* is;

}

ostream &write(ostream &os) *override* {

IoBase::write(os); **// in general, punem la inceputul apelul**

os<< "player1: " << player1;

os<< " player2: " << player2;

*return* os;

}

};

**Pasul 4 - acesta este important pentru rezultatul final**

**Vom adăuga încă doua funcții în clasa de baza, astfel încât în main() vom putea citi tot cu ajutorul** cin >> **si**  cout << **.**

***class* IoBase {**

***private*:**

***public*:**

***// ...***

***friend* ostream &*operator*<<(ostream &os, *const* IoBase &base) {**

***return* base.write(os);**

**}**

***friend* istream &*operator*>>(istream &is, IoBase &base) {**

***return* base.read(is);**

**}**

**};**

### ***Exercitiu (Bonus 1)***

Se cere sa utilizati in clasele PersoanaEvaluata, Pupil, Student, CourseTaker afisarea si citirea, folosind metoda prezentata mai sus.

### 

Puteți folosi de acum încolo această clasă în toate exercițiile și proiectele.

## >> și << implementate automat pentru vector<OriceClasa>

M Mai avem inca un truc pentru a putea implementa >> chiar mai rapid:

Crearea unei functii de tip template care va putea citi orice tip de vectori.

Va „daruiesc” un exemplu de functie template, dupa care puteti lucra in a crea o alta functie template care sa ne permita sa putem rula orice cod de genul:

vector<Student> students;

cin >> students;

### ***Exemplu template functions***

*template*<*typename* T> *// T va fi tipul generic, adica la rulare add(3,2) determina T->int, la rulare add(string(“a”),string(“b”)) determina T->string etc.*

T add(T a, T b){

*return* a + b;

}

*int* main() {

cout << add(2,3) << '\n'; *// afiseaza 5*

cout << add(2.3,3.1) << '\n'; *// afiseaza 5.4*

cout << add(string("234"), string("5")) << '\n'; *// afiseaza 2345*

}

Cel mai bine sunt sincer. Nu există o grămadă de locuri unde folosim funcții template, dar e important să știm de existența lor deoarece ele apar și în Java, C# și tot ele sunt folosite în implementarea containerele STL (vector, map, queue etc). Deci sunt utile, chiar daca nu le vom folosi mereu.

### ***Exercițiu util pentru templates***

(veti putea refolosi aceasta functie in orice alte exercitii)

Supraîncărcați global o funcție de tip template care primeste un vector STL format din elemente de tip T (T fiind tipul generic) care va citi și introduce n elemente de tip T în vectorul nostru. Aveți grijă să goliți vectorul. N-ul va fi citit tot de la consolă.

Hint: antetul va fi similar cu:

istream& *operator*>>(istream& is, vector<string> v);

Dar în loc să folosim **string** vom folosi un tip generic și veți putea rula în main cod precum:

*int* main() {

vector<string> vs;

vector<Pupil> vp;

cin >> vs;

cin >> vp;

cout << vs;

cout << vp;

}

BONUS 2: implementati si testati si un operator pentru afisare vector, care afiseaza toate elementele, separate prin spatiu ‘ ‘.