## Programare orientată obiect

## Obiective

- **Cunoașterea şi înțelegerea conceptelor specifice programării orientate obiect**

- **Abilități de programare în limbajele de programare C și C++**

## Obiectivele specifice:

• Scrierea de programe de scară mică/mijlocie cu interfețe grafice utilizator folosind C++ și QT.
• Proiectarea orientată obiect pentru programe de scară mică/mijlocie
• Explicarea/Înțelegerea structurilor de tip clasă ca fiind componente fundamentale in construirea aplicațiilor.
• Înțelegerea rolului moștenirii, polimorfismului, legării dinamice și a structurilor generice în realizarea codului reutilizabil.
• Utilizarea claselor/modulelor scrise de alți programatori în dezvoltarea sistemelor proprii
• Folosirea de instrumente: debugger,  memory leak detector, code coverage, static code analyser, profiler

**1 Elemente de bază ale limbajului C.**

- Elemente de bază ale limbajului C.  Elemente lexicale. Operatori. Conversii.
- Tipuri de date. Variabile. Constante. Domeniul de vizibilitate și durata de viață
- Declararea și definirea funcțiilor.

**2-3. Programare modulară în C/C++. Tipuri de date**

- Funcții. Parametri.
- Fișiere header. Biblioteci. Implementarea modulară a TAD-urilor.
- Tipuri de date derivate și tipuri definite de  utilizator, alocare dinamică în C++.
- Tipuri de date: vectori și structuri, pointeri și referințe.
- Gestiunea memorie in C/C++. Pointeri la funcții și pointeri spre void.

**4. Programare orientată obiect în C++.**

- Clase și obiecte. Membrii unei clase. Modificatori de acces. Constructori/destructori.
- Gestiunea memoriei in C++ (RAII)
- Implementarea TAD-urilor in C++
- Diagrame UML pentru clase (membri, acces).

**5. Elemente de programare generica**

- Functii/clase parametrizate. Mecanismul de template din C++
- Implementare TAD-uri folosind clase parametrizate
- Containere şi iteratori – biblioteca STL

**6-8 Moștenire / Polimorfism / Ierarhii de clase**

- Moștenire simplă. Clase derivate. Principiul substituției.
- Supraîncărcarea metodelor. Moștenire multiplă. Relații de specializare/generalizare
- Mostenire, polimorfism
- Stream-uri I/O. Ierarhia de clase I/O. Formatare. Manipulatori. Fișiere text.
- Exceptii, spatii de nume. Fișiere text.

**9-11 Interfeţe grafice utilizator  / Elemente de programare bazată pe evenimente**

- QT Toolkit: instalare, instrumente si module Qt. Componente grafice utilizator. Layout management. Proiectare GUI.
- Evenimente: Semnale si sloturi Qt. Callback/Observer
- Componente grafice cu modele. Șabolul MVC.
- Studiu de caz. Detalii comenzi – Produse.

**12-13 Șabloane de proiectare.**

- Șablonul Observer
- Șabloane de proiectare Façade, Strategy.
- Șablonul de proiectare Composite

**Note/Reguli**

**Calcului notei:**

Notă Laborator 30% (media notelor de laborator)
Notă Simulare 10%
Notă examen scris 30%
Notă examen practic 30%

Pentru promovare este nevoie de **minim 5** la fiecare notă in afară de simulare.

**Laborator/prezențe**

Cei care nu satisfac criteriu cu numărul de prezență la laborator/seminar nu participa la examen (au picat materia).

Cei care nu au nota 5 la laborator nu pot intra in examenul din sesiune. Pot veni la examenul din sesiunea de restanță.

În restanță se predau toate laboratoare (cei care nu au luat minim 5 in timpul semestrului). Nota maxima pentru nota pe activitatea de laborator este 5 in acest caz.

**Restanță**

În sesiunea de restanță se poate da examenul scris, examenul practic sau ambele. Valabil atât pentru cei care au picat examenul (examenele) cât și pentru cei care vin la mărire de notă.

## Reguli / desfășurare

Sa aveți la voi un act de identitate (carnet de student, buletin, pașaport)

Sa aveți toate taxele plătite la zi (daca este cazul)

Se da examenul scris apoi examen practic cu o pauza intre ele

Sa va prezentați la examen înainte cu 15 minute de ora stabilita.

Nu copiați. Daca observați orice tentativa de fraudare / nereguli sa le semnalați in timpul examenului.

## Sugestii

Veniți odihniți la examen.
Aveți la voi o sticla de apa pentru hidratare.
Sa mâncați ceva înainte de examen sau in pauza intre scris si practic.
Nu va stresați excesiv. Tot timpul este o alta șansa sa mai dai examenul. Nu se termina lumea cu un examen ratat.

## Examenul scris

Examenul scris se da în timpul sesiunii.

Durata aproximativ 1.5

Tipuri de probleme:

- sintaxă C++ , algoritmi
  - Se dă o funcție C++ - se cere specificare și testare pentru funcția dată
  - se dă specificația sau niște funcții de test - se cere implementarea c++

- concepte din C/C++ si programarea orientată obiect : alocare dinamică, constructor, destructor, moștenire, polimorfism, metode virtuale, clase abstracte, suprascriere, supraâncărcare, excepții, streamul IO, containere, iteratori, algoritmi STL, etc.
  - Se dă un cod c++ în care se folosesc anumite concepte - se cere rezultatul rulării, identificarea/explicarea erorilor,
- Code guidlines: memory leak, dangling pointer, const correctness, exception safe code.
  - Se da codul c++ trebuie sa identificați problemele si sa scrieți codul echivalent care elimina ne-ajunsurile
- UML – C++ - Șabloane de proiectare
  - se dă un cod c++ - se cere diagrama UML de clasă
  - Se dă o diagramă UML de clase – se cere codul c++
  - Se dă o descriere a unor clase și relațiile intre clase – se cere codul c++
- Qt
  - se da codul sursă Qt – se cere o schiță a interfeței grafice, explicații despre ce își propune codul dat
  - se da o schiță a interfeței grafice – se cere codul Qt care construiește interfața utilizator conform schiței

## Exemple de probleme examen scris

Specificati si testati functia:

```
vector<int> f(int a) {
	if (a < 0)
		throw MyException{"Illegal argument"};
	vector<int> rez;
	for (int i = 1; i <= a; i++) {
		if (a % i == 0) {
			rez.push_back(i);
		}
	}
	return rez;
}
```

Definiți o clasa *grades* ce reprezintă notele obținute de un student, astfel încât următoarea secvența **C++** sa fie corecta sintactic si sa efectueze ceea ce indica comentariile.

```
#include <iostream>
#include <vector>
int main() {
  grades<int> myg;
  myg = myg + 10; // adaugam nota 10 la OOP
  myg = myg + 9;   //adaugam nota 9 la FP
  double avg = 0.0;
  for (auto g:myg){  //iteram toate notele
     avg+=g;
  }
  return avg/myg.getNRGrages();//compute average
}
```

Indicați rezultatul execuției pentru următorul program c++. Daca sunt erori indicați locul unde apare eroarea si motivul.

```
#include <iostream>
using namespace std;
class A {
public:
 A() {cout << "A" << endl;}
 ~A() {cout << "~A" << endl;    }
 void print() {cout << "print" << endl;}
};
void f() {
	A a[2];
	a[1].print();
}
int main() {
	f();
}
```

Code guidlines – se da un cod C++ care funcționează (compilează) dar care nu respecta stilul promovat de noi sau are probleme cu: memoria, copieri ne-necesare, const correctnes, etc.

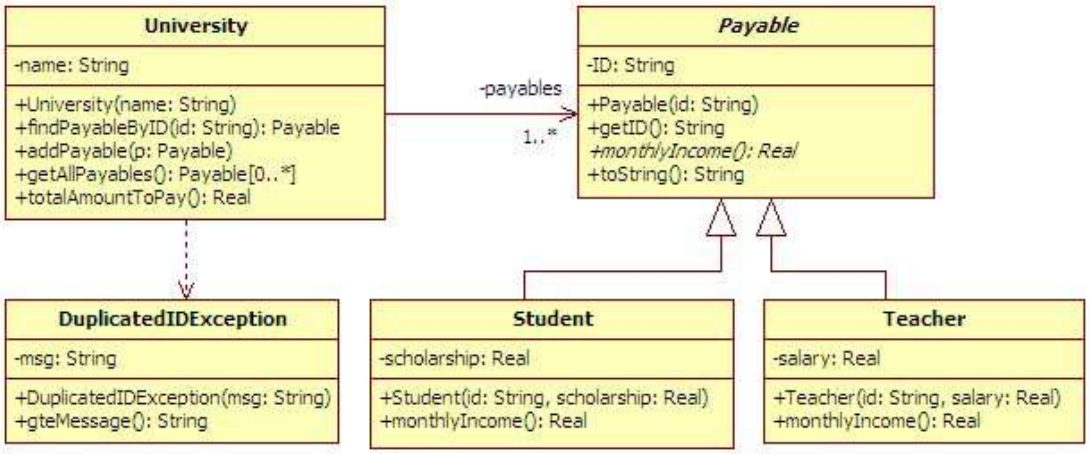
Identificați problemele in codul C++ de mai jos. Scrieți o funcție echivalent funcțional care rezolva problemele identificate.

```
int function(vector<Point> points) {
	Point* aux = new Point{ 0,0};
	for (auto p: points) {
		aux->x += calcul(p.x);
		aux->y += calcul(p.y);
	}
	if (aux->x > 10 || aux->y > 10) {
		return;
	}
	int rez = aux->x + aux->y;
	delete aux;
	return rez;
}
```

```
//const correctness -se face o copie la vector care nu e necesara
int function(vector<Point> points) {
	//Now owning raw pointers - de evitat pointerii cand nu e necesar
	Point* aux = new Point{ 0,0 };
	//const correctness - se face o copie la fiecare point
	for (auto p : points) {
//exception safe code - daca calcul arunca exceptie avem memory leak
		aux->x += calcul(p.x);
		aux->y += calcul(p.y);
	}
	if (aux->x > 10 || aux->y > 10) {
		return;//memory leak
	}
	int rez = aux->x + aux->y;
	delete aux;
	return rez;
}
```

```
int myfunction(const vector<Point>& points) {
	Point aux{ 0,0 };
	for (const auto& p : points) {
		aux.x += calcul(p.x);
		aux.y += calcul(p.y);
	}
	if (aux.x > 10 || aux.y > 10) {
		return;
	}
	return aux.x + aux.y;
}
```

Scrieți codul C++ ce corespunde diagramei de clase UML.




- Universitatea are doua tipuri de entitati: studenti si profesori.
- Pentru studenti universitatea plateste burse lunare, pentru profesor trebuie sa plateasca salarii. Metoda *monthlyIncome()* returneaza suma datorata pentru fiecare entitate (valoarea bursei pentru student, respectiv salarul pentru profesor).
- Metoda *toString()* din clasa *Payable* tipareste id-ul, urmat de suma de platit.
- Metoda *getAmountToPay()* din clasa *University* calculeaza suma totala de platit (atat burse cat si salarii).
- Metoda addPlayables arunca exceptie daca se adauga un Playable cu un id care mai exista

Scrieți un program care creează o instanța de Universitate, adaugă mai multe entități (atât studenți cat si profesori) si tipărește toate entitățile (folosind metoda toString) si suma totala ce trebuie plătit de către universitate (getAmountToPlay()).

Implementați corect gestiunea memoriei. Exemplificați aruncarea si tratarea excepțiilor in C++ prin adăugări a doua entități cu același id.

## Examen practic

În aceeași zi cu examenul scris

Durată: 2.5 - 3 ore

Se cere o aplicație cu interfață grafică utilizator (QT).

Aplicația se dezvoltă pornind de la un proiect gol, nu se pot folosi coduri surse externe (existente)

Se poate folosi:

- QT Assistant
- Pe o foaie A4 se pot scrie API - signaturi de metode, constante, operatori, include-uri,etc (fără algoritmi)

Aplicația de dezvoltat:

- Citește/scrie date din/in fișier text
- Validează datele introduse de utilizator
- Folosește arhitectura stratificată
- Specificații și teste
- 4-6 funcționalități
  - Se punctează doar acele funcționalități care se pot demonstra executând aplicația (nu se dau puncte pentru cod sursă)

Se pot folosi laptopuri proprii la examen - orice mediu de dezvoltare

Se pot folosi calculatoarele facultății - Qt + Qt Creator

Ca si la simulare, in plus posibil:

- Mai multe ferestre, componente create dinamic
- Observer
- Componente cu modele (Model/View)
- Desenare (QPainter)

**Examen**

Nu copiați. Copiatul este furt/frauda.

Cei care copiază pica materia (nu sunt primiți in examenul din restante)

Pentru pregătire - faceți exerciții în condiții similare ca și la simulare. Vedeți cât timp vă ia, care sunt problemele de care vă loviți.

Construiți aplicația incremental. Pași mici, salvat-compilat-testat frecvent. Folosiți dezvoltarea bazate pe teste.

Feature driven – sa va concentrați pe o singura cetinița si implementați minimul necesar pentru respectiva funcționalitate

Adăugați câte o metodă odată să puteți reveni ușor la o versiune anterioară care funcționa.

Nu ignorați erorile, rezolvați problema înainte să treceți mai departe. Nu treceți mai departe dacă nu știți dacă ultimul lucru adăugat funcționează.

Nu ignorați warningurile, ele pot indica erori în program (ex. pointeri)

Folosiți containere , algoritmi STL.

Nu implementați lucruri care nu se cer, ele nu se vor puncta (Ex nu implementați ștergere dacă nu se cere în problemă). Nu se dau puncte pe volumul de cod sursă.

Dacă ceva nu va iese, încercați sa treceți peste, implementați o variantă banală (Ex. În loc să citească din fișier returnează niște obiecte hardcodate) și reveniți ulterior pentru a rezolva problema.

Dacă folosiți laptopul propriu asigurați-vă ca funcționează (încărcător, softurile necesare instalate şi  funcționale).