Curs 6 Gestiunea memoriei in C++

- **Alocare dinamica. Destructor. RAII. Rule of three.**
- **Tratarea excepțiilor – exception safe code**
- **Moștenire**

Curs 5

- Template (Programare generica)
- STL – Standard Template Library
- Tratarea excepțiilor in C++

Alocare dinamică de obiecte

operatorul new alocă memorie pe heap si inițializează obiectul (apelând constructorul clasei)

```
    Rational *p1 = new Rational;
    Rational *p2 = new Rational{2, 5};
    cout << p1->toFloat() << endl;
    cout << (*p2).toFloat() << endl;
    delete p1;
    delete p2;
```

Orice variabilă creată cu **new** trebuie distrusă cu **delete**. (fiecare **new** exact un **delete**). Programatorul este responsabil cu eliberarea memoriei

Pentru a distruge vectori statici se folosește **delete** []

```
char* nume = new char[20];
delete[] nume;
```

```
//se apeleaza constructorul fara parametrii de 10 ori
Pet* pets = new Pet[10];
delete[] pets;
```

De preferat sa nu se folosească malloc/free si **new**/ **delete** in același program, in special sa distrugă cu free obiecte alocate cu **new**

new alocă memorie si apelează constructorul clasei

Destructor

O metoda speciala a clasei.
Destructorul este apelat de fiecare data când se dealocă un obiect

- dacă am alocat pe heap (**new**), se apelează destructorul când apelez **delete/delete[]**
- dacă e variabilă alocat static, se dealoca în momentul în care nu mai e vizibil (out of scope)

```
DynamicArray::DynamicArray()
{
  cap = 10;
  elems = new Rational[cap];
  size = 0;
}
```

```
DynamicArray::~DynamicArray()
{
    delete[] elems;
}
```

Gestiunea memoriei in C++

Destructorul este apelat:

- Dacă obiectul a fost creat cu new → când apelam delete
- Dacă a fost alocat pe stack -> cand părăsește domeniul de vizibilitate (out of scope)
- Implementați destructor si eliberați memoria alocata cu new !!!!

Constructorul este apelat:

- când declarăm o variabilă pe stack
- Dacă creăm o variabila cu new (pe heap)
- Dacă creăm o copie a obiectului (copy constructor/assignment operator)
  - atribuire
  - transmitere parametrii prin valoare
  - returnam obiect prin valoare dintr-o funcție
- implementați constructor de copiere si supraîncărcați operatorul = !!!

Cum gestionam memoria in C++ (RAII)

Orice memorie alocată pe heap (new) ar trebui încapsulată într-o clasa (in general ne referim la o astfel de clasă clasă Handler)

Ar trebui sa alocăm memoria in constructor si să dealocăm memoria in destructor. Astfel memoria alocata este strâns legat de ciclul de viată a obiectului Handler
Când cream obiectul se alocă memoria, când obiectul este distrus se eliberează memoria.

RAII (Resource Acquisition Is Initialization) este o tehnică prin care legam ciclul de viată a unei resurse (memorie, thread, socket, fisier, mutex, conexiune la baza de data) de ciclul de viață a unui obiect.

Folosind RAII in programele C++:

gestiunea memoriei se simplifică – destructor / constructor / constructor de copiere sunt apelate automat

putem controla cat timp este memoria ocupată/ când este dealocată (are același viată ca si domeniul de vizibilitate a obiectului care încapsulează resursa)

Clase – încapsulare, abstractizare

clasele ajuta sa raționăm asupra codului local, la un nivel de abstractizare mai mare

```
void someFunction(Pet p){
…
}
```

```
Pet someFunction(int a){
…
}
```

Parametrul p este transmis prin valoare

Daca `Pet` este un struct din C, trebuie sa inspectam definiția structului Pet pentru a decide dacă a transmite prin valoare este sau nu o abordare legitimă

Daca `Pet` este o clasa scrisa bine (rules of three) codul funcționează corect (nu am nevoie sa știu detalii despre clasa – lucrez la un nivel de abstractizare mai mare, clasa încapsulează/ascunde detaliile de implementare

```
//vers 1
typedef struct {
      char name[20];
      int age;
} Pet;
//a simple copy (bitwise) will do
```

```
//vers 2
typedef struct {
      char* name;
      int age;
} Pet;
// when we make a copy we need to
take care of the name field
```

Dacă am un struct C folosirea pointerilor nu rezolva dilema.

In acest caz nu trebuie sa mă gândesc la copiere dar trebuie sa mă gândesc la responsabilitatea dealocării memoriei.(-> este nevoie sa mă uit la implementarea funcției **someFunction** nu pot sa ignor detaliile de implementare si sa raționez local asupra codului)

```
…
Pet* p = someFunction(7);
//do i need to delete/free p?
//need to inspect someFunction
```

```
Pet* someFunction(int a){
…
}
```

Gestiunea memoriei alocate dinamic C vs C++

C	C++

`Pet p; // p este neinițializat`	`Pet p; // p este inițializat,`
	`// se apelează constructorul`

Funcție pentru creare, distrugere Trebuie urmărit in cod folosirea corectă a acestor funcții (este ușor sa uiți să apelezi)	Constructor / Destructor **Sunt apelate automat de compilator**
Funcție care copiază Trebuie urmărit/decis momentul in care dorim sa facem copie	Constructor de copiere, operator de assignment **Sunt apelate automat de compilator**
Pointeri peste tot in aplicație Este greu de decis cine este responsabil cu alocarea de-alocarea	Pointerii se încapsula într-o clasă handler Gestiunea memoriei este încapsulat într-o clasa (**RAII**). Ciclu de viată pentru memorie este strâns legat de ciclu de viată a obiectului Exista clase handler predefinite ex: `unique_ptr`
Se folosesc pointeri doar pentru a evita copierea Nu este clar dacă/când se dealocă	**Tipul referința** oferă o metodă mult mai transparentă pentru a evita copierea unde nu este necesar
Se compune greu Daca am o lista (care alocă dinamic) cu elemente alocate dinamic (poate chiar alta listă) este relativ greu de implementat logica de dealocare	Se compune ușor Daca am o clasa cu atribute, destructorul obiectului apelează si destructorul atributelor.
Gestiunea memoriei afectează toată aplicația.	**Încapsulare/ascunderea detaliilor de implementare** Modificările se fac doar in clasa

Rule of three

```
void testCopy() {
	DynamicArray ar1;
	ar1.addE(3);
	DynamicArray ar2 = ar1;
	ar2.addE(3);
	ar2.set(0, -1);
	printElems(&ar2);
	printElems(&ar1);
}
```

Daca o clasa este responsabila de gestiunea unei resurse (heap memory, fisier, etc) trebuie sa definim obligatoriu:

- copy constructor

```
DynamicArray::DynamicArray(const DynamicArray& d) {
	this->capacity = d.capacity;
	this->size = d.size;
	this->elems = new TElem[capacity];
	for (int i = 0; i < d.size; i++) {
		this->elems[i] = d.elems[i];
	}
}
```

- Assignment operator

```
DynamicArray& DynamicArray::operator=(const DynamicArray& ot) {
	if (this == &ot) {
		return *this;// protect against self-assignment (a = a)
	}
	delete[] this->elems;	//delete the allocated memory
	this->elems = new TElem[ot.capacity];
	for (int i = 0; i < ot.size; i++) {
		this->elems[i] = ot.elems[i];
	}
	this->capacity = ot.capacity;
	this->size = ot.size;
	return *this;
}
```

- Destructor

```
DynamicArray::~DynamicArray() {
	delete[] elems;
}
```

Review PetStore varianta OOP

Pentru o gestiune corecta a memoriei: Clasele care aloca memorie trebuie sa implementeze:

constructor de copiere – copierea valorilor atributelor in noul obiect
destructor – eliberare memorie
operator = - eliberare memorie valori existente, copierea valorilor atributelor

Valabil pentru orice clasa care este responsabil de gestiunea unei resurse.

TAD VectorDinamic

- are operațiile listei – metodele publice din clasa – interfața clasei (.h)
- detaliile de implementare sunt ascunse – specificații abstracte, implementări in cpp
- elementele sunt obiecte nu pointeri – simplifica gestiunea memoriei

Clasele care nu aloca memorie pot folosi constructor/ destructor/ copy-constructor / assignment default (oferite implicit de compilator)

Daca dorim sa evitam anumite operații (ex. Din motive de performanta) putem folosi

```
PetController(const PetController& ot) = delete;//nu vreau sa se copieze controller
```

In afara de clasa Pet si clasa VectorDinamic nu e nevoie sa folosim pointeri si alocare dinamica.

Gestiunea memoriei se simplifica, se folosește RAII

obiectele au un ciclu de viață bine delimitat (domeniul de vizibilitate)
la creare se apelează constructor, la ieșire din domeniul de vizibilitate se apelează destructor, când folosim transmitere prin valoare sau return prin valoare se apelează copy constructor
Avantaje: creste nivelul de abstractizare: codul se simplifica, gestiunea memoriei se simplifica, folosirea clasei VectorDinamic nu necesita înțelegerea detaliilor de implementare
Dezavantajul abordării: Se fac multe copieri de obiecte
Masuri pentru a evita aceste copieri:
Transmitere prin referința, const correctness
Move Constructor, Move assignment (discutam la următoarele cursuri), Rule of five

Exception-safe code

Dezavantajele folosirii excepțiilor:
Dacă scriem cod care folosește excepții, ar trebui să luăm in considerare apariția unei excepții oriunde in cod.

Este greu să scrii cod care se comportă predictibil (fără buguri) chiar si atunci când apare o excepție. (nu rezultă probleme cu memorie - leak/dangling - sau cu alte resurse)

```
void g() {
	...
}
void f() {
	char* s = new char[10];
	...
	g();//daca g arunca exceptie avem memory leak (nu se mai ajunge la delete s)
	...
	delete[] s;
}
```

Obs: astfel probleme sunt generale (chiar dacă nu folosim excepții) – puteam avea un simplu return înainte de delete si rezulta memory leak
Ce înseamnă exception-safe code:
Basic: chiar dacă apare o excepție:
- invarianții rămân valabili (obiectul nu ajunge într-o stare inconsistentă)
- nu avem resource leak

Strong:
- apariția unei excepții nu are nici un efect vizibil
- operația ori se face in totalitate ori nu se modifica nimic (tranzacție)

Exception-safe code:

Pentru orice resursă pe care o gestionam (ex. Memorie) creăm o clasă

Orice pointer o sa fie încapsulat intr-un obiect, obiect automat - gestionat de compilator, declarat local in funcție (nu creat pe heap cu new) - No raw pointer

Ne folosim de RAII – beneficiem de faptul ca destructorul se apelează când execuția părăsește scop-ul local (chiar daca se iese aruncând o excepție)

Facem asa:

```
void f() {
	A a{ "asda",10 };
	...
	g();//daca g arunca excepție destructorul lui A se apelează
	...
}
```

In loc de:

```
void f() {
	A* a = new A{ "asda",10 };
	...
	g();//daca g arunca excepție avem memory leak (nu se mai ajunge la delete s)
	...
	delete a;
}
```

Unde chiar avem nevoie de pointeri si alocare pe heap putem folosi **unique_ptr**

```
#include <memory>
using std::unique_ptr;
void f() {
	 unique_ptr<char[]> ptr_s = std::make_unique<char[]>( 10 );
	...
	g();//daca g arunca excepție destructorul lui A se apelează
	...
	//când ieșim din funcție (excepție sau normal) destructorul lui ptr_s
	// apelează delete[] pentru char* de 10 caractere alocate pe heap
}

void f2() {
	auto ptr_s= make_unique<A>("asda",10);
	//când iesim din funcție destructorul lui ptr_s apelează delete
	// pentru obiectul A create pe heap de metoda make_unique
}
```

unique_ptr – smart pointer

Clasa care conține un pointer, la apelul destructorului eliberează memoria ocupata de obiectul referit (face delete)

Util pentru a gestiona corect memoria, unique_ptr modelează „unique ownership”

```
int* f() {
.....
}

int main() {
  int* pi = f();
  //trebuie sa dealoc pi? cine este responsabil cu dealocarea?
.....
  return 0;
}
```

```
#include <memory>
std::unique_ptr<int> f() {
      .....
}
int main() {
      std::unique_ptr<int> pi = f();
      //sunt responsabil cu dealocarea, se va dealoca automat cand pi
iese din scope
.....
  return 0;
}
```

Regula: No raw pointer - No raw owning pointer

De evitat folosirea de pointeri (naked pointer) – folosirea excesiva de pointer face (aproape) imposibila gestiunea corecta a memoriei.

Moștenire

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

Dacă A și B sunt două clase unde B moștenește 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 adaugă 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 acces

Definesc reguli de acces 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 accesul 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 constructorul din clasa de baza. Sa ne asigurăm ca obiectul este inițializat 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 constructorul din clasa de bază, se apelează automat constructorul implicit
- Dacă nu exista constructor implicit se generează 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";
}
```

Inițializare.

Când definim constructorul putem inițializa 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 membre 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.

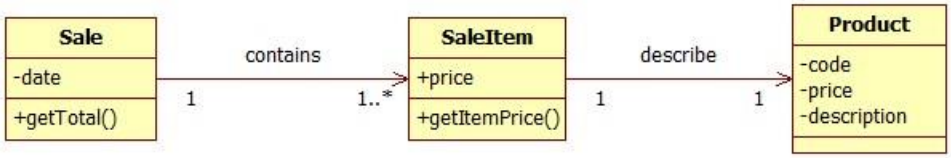
Un obiect de tipul clasei derivate se poate folosi în orice loc (context) unde se cere un obiect de tipul clasei de bază. (upcast implicit!)

```
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

Relația de asociere UML (Associations): Descriu o relație de dependență structurală între clase

Elemente posibile:

- nume
- multiplicitate
- nume rol
- uni sau bidirecțional

***Tipuri de relaţii de asociere**

- Asociere
- Agregare (compoziție) (whole-part relation)
- Dependență
- Moștenire

Are (has a):

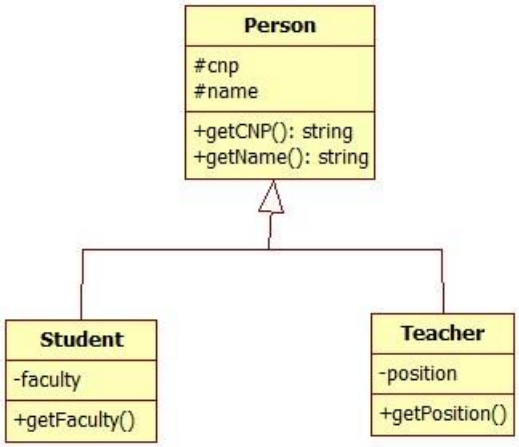
- Orice obiect de tip A are un obiect B.
- SaleItem are un Product. Persoana are nume (string)
- in cod apare ca şi o variabilă membră

Este ca şi (is a ,is like a):

- Orice instanță de tip A este şi de tip B
- Orice student este o persoană
- se implementează folosind moștenirea

Relația de specializare/generalizare – Reprezentarea UML .

Folosind moștenirea putem defini ierarhii 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