Alocare dinamica de obiecte

Operatorul new aloca memorie pe heal si initializeaza obiectul apeland 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 variabila creata cu new trebuie distrusa cu delete (fiecare new are exact un delete).
Programatorul este responsabil cu eliberarea memoriei.
Pentru a distruge vectori statici se foloseste 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 foloseasca malloc/free si new/delete in acelasi program, in special sa distruga cu free obiecte alocate cu new.

Destructor

Este o metoda speciala a clasei. Destructorul este apelat de fiecare data cand se dealoca un obiect. Daca am alocat pe heap (new), se apeleaza destructorul cand apelez delete/delete[]. Daca variabila este alocata static, se dealoca in momentul in care nu mai este vizibila.

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

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

Gestiunea memoriei in C++

Destructorul este apelat :

- Daca obiectul a fost creat cu new, cand apelam delete.
- Daca a fost alocat pe stack, cand paraseste domeniul de vizibilitate (out of scope).

Constructorul este apelat :

- Cand declaram o variabila pe stack.
- Daca cream o variabila cu new (pe heap).
- Daca cream o copie a obiectului (copy constructor/assignment operator)
  - atribuire
  - transmitere parametrii prin valoare
  - returnam obiect prialn voare dintr-o functie

Clase - incapsulare, abstractizare

Clasele ajuta sa rationam asupra codului, 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 definitia structului Pet pentru a decide daca a transmite prin valoare este sau nu o abordare corecta.

Daca Pet este o clasa codul functioneaza corect (nu am nevoie sa stiu detalii despre clasa -

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

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

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

Pentru o gestiune corecta a memoriei, clasele care aloca memorie trebuie sa implementeze obligatoriu :

- Copy constructor

```c
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

```c
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;
}
```

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

Daca dorim sa evitam anumite operatii putem sa folosim :

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

Exception-safe code

Dezavantajele folosirii exceptiilor :

Daca scriem cod care foloseste exceptii ar trebui sa luam in considerare aparitia unei exceptii oriunde in cod. Este greu sa scriem cod care se comporta predictibil fara memory leak.

```
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;
}
```

Observatie : astfel de probleme sunt generale, chiar daca nu folosim exceptii - am putea sa avem un simplu return inainte de delete si rezulta memory leak.

Exception-safe code se bazeaza pe faptul ca chiar daca apare o exceptie, nu avem resource leak si invariantii raman valabili, adica nu se ajunge intr-o stare inconsistenta. Acest lucru se poate aplica creand o clasa pentru orice resursa pe care o gestionam (memoria). Orice pointer incapsulat intr-un obiect este gestionat de compilator, declarat local in functie (nu creat pe heap cu new). Beneficiem de faptul ca destructorul se apeleaza cand executia paraseste scop-ul local (inclusiv cand se iese aruncand o exceptie).

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 sunt necesari pointerii si alocarea pe heal putem sa ne folosim de 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

Este clasa care contine un pointer pentru care la apelul destructorului se elibereaza memoria ocupata de obiectul referit (se face delete).

```
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;
}
```

Mostenire

Mostenirea permite definirea de clase noi (clase derivate) reutilizand clase existente (clase de baza). Clasa noua creata mosteneste comportamentul (metode) si caracteristici (variabile membre, starea) de la clasa de baza.

Daca A si B sunt doua clase unde B mosteneste 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 langa cele mostenite 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;
};
```

Mostenire simpla. Clase derivate.

Daca clasa B mosteneste de la clasa A atunci :

- orice obiect de tip B are toate variabilele membre din clasa A
- functiile din clasa A pot fi aplicate si asupra obiectelor de tip B (daca vizibilitatea permite)
- clasa B poate adauga variabile membre si/sau metode pe langa cele mostenite din A

```
class A:public B{
….
}
```

clasa B = Clasa de baza (superclass, base class, parent class)
clasa A = Clasa derivata (subclass, derived class, descendent class)
membrii (metode, variabile) mosteniti = membrii definiti in clasa A si nemodificati in clasa B
membrii redefiniti (overriden) = definit in A si in B (in B se creeaza o noua definitie)
membrii adaugati = definiti doar in B

Vizibilitatea membrilor mosteniti

Daca clasa A este derivata din clasa B :

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

```
class A:public B{
….
}
```

public : membrii publici din clasa B sunt publici si in clasa A

```
class A:private B{
…
}
```

private : membrii publici din clasa B sunt privati in clasa A

```
class A:protected B{
…
}
```

protected : membrii publici din clasa B sunt protejati in clasa A (protejati, adica se vad doar in clasa A si in clase derivate din A)

Modificatori de acces

Definesc reguli de acces la variabile membre si metode dintr-o clasa

public : poate fi accesat de oriunde

private : poate fi accesat doar in interiorul clasei

protected : poate fi accesat in interiorul clasei si in clasele derivate

protected se comporta ca si 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 in clase derivate

- Constructorii si destructorii nu sunt mosteniti
- Constructorul din clasa derivata trebuie sa apeleze constructorul din clasa de baza
- Similar si pentru destructor, trebuie sa ne asiguram ca resursele gestionate de clasa de baza sunt eliberate

- Daca nu apelam explicit constructorul din clasa de baza, se va apela automat constructorul implicit, daca acesta nu exista se genereaza o eroare

Explicit :

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

Implicit :

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

Initializare

Cand definim constructorul putem initializa variabilel membre chiar inainte sa se execute corpul constructorului astfel :

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

Initializare clasa de baza

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

Apelare metoda din clasa de baza

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

Creare / distrugere de obiecte (clase derivate)

Creare

- se aloca suficienta memorie pentru variabilele membre din clasa de vaza
- se aloca suficienta memorie pentru variabilele membre noi din clasa derivata
- se executa constructorul clasei de baza pentru initializarea atributelor din clasa de baza
- se executa constructorul din clasa derivata pentru initializarea atributelor din clasa derivata

Distrugere

- se apeleaza destructorul din clasa derivata
- se apeleaza destructorul din clasa de baza

Principiul substitutiei

Un obiect de tipul clasei derivate se poate folosi in orice context unde se cere un obiect de tipul clasei de baza (upcast)

```
      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
```