SUBIECTE TEORETICE EXAMEN OOP

-constructori

-Ordinea apelarii constructorilor

* Daca avem constructorul de copiere dat de compilator: c. c. Baza
  c. c. Derivata
* Daca avem constructor de copiere scris de programator: c. initializare Baza
  c. copiere Derivata
* Daca avem constructor de copiere scris de programator: apelam explicit c. copiere Baza
  c. copiere Derivata

-ordinea -intai constructorul BAZEI (daca mosteneste o clasa) apoi constructorul MEMBRILOR si apoi constructorul clasei -indiferent de ordinea din lista de apel explicit

-initializare automata

-nu pot intoarce valori (nu au tip de intoarcere)

-au numele clasei

-chemati de fiecare data cand un obiect de acel tip este creat

-in ordinea in care sunt create obiectele

-constructori parametrizati

-overload

-constructori de copiere

-classname (const classname &o) {
// body of constructor
}

-este folosit DOAR la copiere, obiectul e deja initializat : myclass ob = myclass(3, 4);

-myclass A=B;

Problema apare la alocare dinamica de memorie: A si B folosesc aceeasi zona de memorie pentru ca pointerii arata in acelasi loc . Destructorul lui MyClass elibereaza aceeasi zona de memorie de doua ori (distruge A si B)

-constructor cu un parametru

-exemplu pentru o clasa X care are constructor X(int i): X ob=99; sau X ob(99);

-polimorfism pe constructori

-nu se pot initializa obiecte dintr-o lista alocata dinamic

-Pentru array alocate dinamic trebuie sa existe constructorul fara parametrii

-destructori

-dupa ce main s-a terminat, in ordinea inversa a constructorilor

-chemati de fiecare data cand un obiect de acel tip este distrus

-executa operatii cand obiectul nu mai este folositor

-memory leak

- ~numele_clase() {..}

-constructorii si destructorii nu se mostenesc, trebuie redefiniti

-clase

- mentioneaza proprietatile generale ale obiectelor din clasa respectiva
- clasele nu se pot “rula”
- folositoare la encapsulare (ascunderea informatiei)
- reutilizare de cod: mostenire

-specificatori de acces (private, protected, public)

-pentru definirea fiecarei functii se foloseste operatorul de rezolutie de scop ::

-functii prieten

- cuvantul: friend
- pentru accesarea campurilor protected, private din alta clasa
- folositoare la overload-area operatorilor, pentru unele functii de I/O, si portiuni de cod interconectate, in rest nu se prea folosesc

- functii prieten din alte obiecte

-clase prieten

-daca avem o clasa prieten, toate functiile membre ale clasei prieten au acces la membrii privati ai clasei

-functii inline

- foarte comune in clase
- doua tipuri: explicit (inline) si implicit
- executie rapida
- este o sugestie/cerere pentru compilator
- pentru functii foarte mici

-memorie limitata, nu putem declara toate functiile inline

-functii care intorc obiecte

- o functie poate intoarce obiecte
- un obiect temporar este creat automat pentru a tine informatiile din obiectul de intors
- acesta este obiectul care este intors
- dupa ce valoarea a fost intoarsa, acest obiect este distrus
- probleme cu memoria dinamica: solutie polimorfism pe = si pe constructorul de copiere

-supraincarcarea functiilor

-acelasi nume pentru functii diferite

-compilatorul foloseste tipul si numarul de parametrii pentru diferentiere

-daca diferenta este doar in tipul de date intors: eroare la compilare

-argumente implicite

-se specifica o singura data, pot fi mai multe, toate sunt la dreapta

- putem defini valori implicite pentru parametrii unei functii
- valorile implicite sunt folosite atunci cand acei parametrii nu sunt dati la apel

-encapsulare

- date si metode impreuna

- urmatorul cod nu poate fi folosit in main() : stack1.tos = 0; // Error, tos is private

Polimorfism la compilare: ex. max(int), max(float)

Polimorfism la executie: RTTI

-supraincarcarea operatorilor

* **functii operator membri ai clasei**

-in interiorul clasei : ret-type operator# (arg-list) { ... }

-ret-type class-name::operator#(arg-list)

{

// operations

}

-# este operatorul supraincarcat (+ - * / ++ -- = , etc.)

-exemplu pentru + : ret-type operator+(const class_name &ob) {...}

-de obicei ret-type este tipul clasei, dar avem flexibilitate

-pentru operatori unari arg-list este vida

-pentru operatori binari: arg-list contine un element

RESTRICTII: -nu se poate redefini si precedenta operatorilor

-nu se poate redefini numarul de operanzi

-rezonabil pentru ca redefinim pentru lizibilitate

-putem ignora un operand daca vrem

-nu putem avea valori implicite; exceptie pentru ()

-nu putem face overload pe . :: .* ?

-e bine sa facem operatiuni apropiate de intelesul operatorilor respectivi

-mostenire: operatorii (mai putin =) sunt mosteniti de clasa derivata

-clasa derivata poate sa isi redefineasca operatorii

* **functii prieten**

-o facem functie prietena pentru a putea accesa rapid campurile protejate
-in functiile prieten nu avem pointerul “this”
-deci vom avea nevoie de toti operanzii ca parametrii pentru functia operator
-primul parametru este operandul din stanga, al doilea parametru este operandul din dreapta
RESTRICTII: -nu se pot supraincarca = () [] sau -> ca functii prieten
-pentru ++ sau -- trebuie sa folosim referinte

```
class class-name
{ int numar;
 public:
   ..
  // overloaded prefix ++ operator
  class-name operator++ ()
   {
   ++numar;  // increment this object
   return class-name(numar);
   }
  // overloaded postfix ++ operator
  class-name operator++( int )
   {
     // save the original value
    class-name T(numar);
    ++numar;    // increment this object
    // return old original value
    return T;
   }
  };
```

DIFERENTE:
-uneori avem insa diferente: pozitia operanzilor
-pentru functii membru operandul din stanga apeleaza functia operator supraincarcata
-daca vrem sa scriem expresie: 100+ob; probleme la compilare=> functii prieten
-in aceste cazuri trebuie sa definim doua functii de supraincarcare: int + tipClasa si tipClasa + int

-supraincarcarea [], ()
-daca avem in clasa : int vector[100];
int &operator[](int i) {return vector[i]; }
- trebuie sa fie functii membru, (nestatice)
- nu pot sa fie functii prieten
- este considerat operator binar
- operatorul [] poate fi folosit si la stanga unei atribuiri (obiectul intors este atunci referinta)

-overload pe ->
-operator unar
-obiect->element
-obiectul genereaza apelul
-elementul trebuie sa fie accesibil
-intoarce un pointer catre un obiect din clasa

-exceptii try-catch
-daca se face throw si nu exista un bloc try din care a fost aruncata exceptia sau o functie apelata dintr-un bloc try: eroare
-daca nu exista un catch care sa fie asociat cu throw-ul respectiv (tipuri de date egale) atunci programul se termina prin terminate()

-terminate() poate sa fie redefinita sa faca altceva
-un catch pentru tipul de baza va fi executat pentru un obiect aruncat de tipul derivat . Ar putea sa se puna catch-ul pe tipul derivat primul si apoi catchul pe tipul de baza
-catch(...) { // catch all exceptions }
-throw; (fara argumente) rearunca exceptia prinsa

-pointeri

-O variabila care tine o adresa din memorie
-Are un tip, compilatorul stie tipul de date catre care se pointeaza
-Operatiile aritmetice tin cont de tipul de date din memorie
-Pointer ++ == pointer+sizeof(tip)
-Definitie: tip *nume_pointer;
-Merge si tip* nume_pointer;

-pointerul this

-orice functie membru are pointerul this (definit ca argument implicit) care arata catre obiectul asociat cu functia respectiva
-(pointer catre obiecte de tipul clasei)
-functiile prieten nu au pointerul this
-functiile statice nu au pointerul this
*Un pointer catre baza poate fi folosit si catre derivata
*Pointerii catre clase derivate se folosesc pentru polimorfism la executie (functii virtuale)

-parametru referinta

-la apel prin valoare se adauga si apel prin referinta la C++
-nu mai e nevoie sa folosim pointeri pentru a simula apel prin referinta, limbajul ne da acest lucru
-sintaxa: in functie punem & inaintea parametrului formal

-referinte catre obiecte

-daca transmitem obiecte prin apel prin referinta la functii nu se mai creeaza noi obiecte temporare, se lucreaza direct pe obiectul transmis ca parametru
-deci copy-constructorul si destructorul nu mai sunt apelate
-la fel si la intoarcerea din functie a unei referinte

-mostenire

-cheama intai constructorul de initializare pentru baza si apoi cel pentru clasa derivata
-daca in clasa derivata se redefineste o functie suprascrisa (overloaded) in baza, toate variantele functiei respective sunt ascunse (nu mai sunt accesibile in derivata)
-putem face overload in clasa derivata pe o functie overloaduita din baza (dar semnatura noii functii trebuie sa fie distincta de cele din baza)

-lista de initializare pentru constructori

-in aceasta lista NU avem voie sa initializam variabilele statice (in constructor avem voie)
-pentru constructorul din clasa derivata care mosteneste pe baza
derivata:: derivata(int i): baza(i) {…}
-spunem astfel ca acest constructor are un param. intreg care e transmis mai departe catre constructorul din baza
-lista de initializare se poate folosi si pentru obiecte incluse in clasa respectiva
-fie un obiect gigi de tip Student in clasa Derivata . Pentru apelarea constructorului pentru gigi cu un param. de initializare i
Derivata::Derivata(int i): Baza(i), gigi(i+3) {…}

-AMBIGUITATE la mostenire multipla
-clasa baza este mostenita de derivata1 si derivata2 iar apoi clasa derivata3 mosteneste pe derivata1 si 2
-in derivata3 avem de doua ori variabilele din baza!!!
-trebuie sa folosim MOSTENIRE VIRTUALA
class derivata: virtual public baza{ … }

-mostenire virtuala

-astfel daca avem mostenire de doua sau mai multe ori dintr-o clasa de baza (fiecare mostenire trebuie sa fie virtuala) atunci compilatorul aloca spatiu pentru o singura copie

-functii virtuale

-definite in clasa baza si redefinite in derivate

-functiile virtuale dau polimorfism

-folosit pentru polimorfism la executie

-daca o functie virtuala nu e redefinita in clasa derivata: cea mai apropiata redefinire este folosita

-upcasting

-clasa derivata poate lua locul clasei de baza

-late binding

-Late binding pentru o functie: se scrie virtual inainte de definirea functiei.

-Pentru clasa de baza: nu se schimba nimic!

-Pentru clasa derivata: late binding insemna ca un obiect derivat folosit in locul obiectului de baza isi va folosi functia sa, nu cea din baza (din cauza de late binding)

-clase abstracte

-Compilatorul da eroare cand incercam sa instantiem o clasa abstracta

-Clasa abstracta=clasa cu cel putin o functie virtuala PURA

-Nu pot fi trimise catre functii (prin valoare)

-Trebuiesc folositi pointeri sau referintele pentru clase abstracte (ca sa putem avea upcasting)

-Daca vrem o functie sa fie comuna pentru toata ierarhia o putem declara virtuala si o si definim.
Apel prin operatorul de rezolutie de scop: cls_abs::functie_pura();

-Putem trece de la func. normale la pure; compilatorul da eroare la clasele care nu au redefinit functia respectiva

-functii virtuale pure

-Functie virtuala urmata de =0
Ex: virtual int pura(int i)=0;

-La mostenire se defineste functia pura si clasa derivata poate fi folosita normal. Daca nu se redefineste functia pura, clasa derivata este si ea clasa abstracta. In felul acesta nu trebuie definita o functie care nu se executa niciodata.

-exemplu de separare dintre interfata si implementare

-overload pe functii virtuale

-Nu e posibil overload prin schimbarea tipului param. de intoarcere (e posibil pentru ne-virtuale)
Nu putem avea in clasa de baza virtual int f(){..} si virtual void f(){..}

-Pentru ca se vrea sa se garanteze ca se poate chema baza prin apelul respectiv, deci daca baza a specificat int ca intoarcere si derivata trebuie sa mentina int la intoarcere

-Exceptie: pointer catre baza intors in baza, pointer catre derivata in derivata

-NU putem avea constructori virtuali

-destructori virtuali

-daca vrem sa eliminam portiuni alocate dinamic si pentru clasa derivata, dar facem upcasting trebuie sa folosim destructori virtuali

-destructori virtuali puri

-Restrictie: trebuiesc definiti in clasa (chiar daca este abstracta)

-La mostenire nu mai trebuiesc redefiniti (se construieste un destructor din oficiu)

```
class AbstractBase {
public:
 virtual ~AbstractBase() = 0;
};
AbstractBase::~AbstractBase() {}

class Derived : public AbstractBase {};
// No overriding of destructor necessary?
```

int main() { Derived d; }

-downcasting -> dynamic_cast

-Folosit in ierarhii polimorfice (cu functii virtuale)

-Static_cast intoarce pointer catre obiectul care satiface cerintele sau 0

-Foloseste tabelele VTABLE pentru determinarea tipului

-dynamic_cast<dest*>(pointer_sursa)

-Daca stim cu siguranta tipul obiectului putem folosi “static_cast”

-template

-In esenta o **functie generica** face auto overload (pentru diverse tipuri de date)

```
template <class Ttype>
ret-type func-name(parameter list)
{
// body of function
}
```

-Specificatia de template trebuie sa fie imediat inaintea definitiei functiei

-In cazul specializarii explicite versiunea sablonului care s-ar fi format in cazul tipului de parametrii respectivi nu se mai creeaza (se foloseste versiunea explicita)

-sintaxa pentru specializare explicita

```
template<>
void nume_functie<tip_date>(tip_date arg1, tip_date arg2,..){}
```

-clase generice

```
template <class Ttype>
class class-name { ... }

class-name <Ttype> ob;
```

-Functiile membru ale unei clase generice sunt si ele generice (in mod automat)

- Nu e necesar sa le specificam cu template

-typename

-Se poate folosi in loc de class in definitia sablonului

-Informeaza compilatorul ca un nume folosit in declaratia template este un tip, nu un obiect

-runtime type identification (RTTI)

-in mod normal ignoram tipul obiectului si lasam functiile virtuale sa proceseze datele

-uneori dorim sa aflam tipul cel mai derivat al unui obiect catre care nu avem decat pointer de tip baza

-se pot face operatii mai eficient cu aceasta informatie

-casting prin referinte, nu exista referinta null, se face catch

-RTTI nu functioneaza cu pointeri void, nu au informatie de tip

-operatorul typeid

-alt fel de a obtine informatie despre tipul obiectului la rulare

-typeid intoarce un obiect type_info

-daca tipul este polimorfic da informatie despre cea mai derivata clasa (tipul dinamic), altfel da informatii statice

-rezultatul unei operatii typeid nu se poate tine intr-un obiect type_info (nu exista constructori accesibili, atribuirea nu e valida)

-se foloseste in comparatii directe

-sirul exact intors depinde de compilator

-typeid(null) genereaza exceptie: bad_typeid

-const si volatile

-obiectele const apeleaza DOAR functii const, obiectele ne-const pot apela atat functii const, cat si functii ne-const

-un obiect const NU poate fi transferat prin referinta ne-const

-idee: sa se elimine comenzile de preprocesor #define

-#define faceau substitutie de valoare
-se poate aplica la pointeri, argumente de functii, param de intoarcere din functii, obiecte, functii membru
-fiecare dintre aceste elemente are o aplicare diferita pentru const, dar sunt in aceeasi idee/filosofie
-trebuie data o valoare pentru elementul constant la declarare
-daca incercam sa schimbam o variabila const primim eroare de compilare
-in C++ compilatorul incearca sa nu creeze spatiu pentru const-uri, daca totusi se transmite catre o functie prin referinta, extern etc, atunci se creeaza spatiu
-const int* u; la fel ca si int const* v;
u este pointer catre un int care este const
-pentru pointeri care nu isi schimba adresa din memorie
int d = 1;
int* const w = &d;
w e un pointer constant care arata catre intregi+initializare
-const pointer catre const element
int d = 1;
const int* const x = &d; // (1)
int const* const x2 = &d; // (2)
-se poate face atribuire de adresa pentru obiect non-const catre un pointer const
-nu se poate face atribuire pe adresa de obiect const catre pointer non-const
int d = 1;
const int e = 2;
int* u = &d; // OK -- d not const
//! int* v = &e; // Illegal -- e const
int* w = (int*)&e; // Legal but bad practice
int main() {} ///:~
-variabilele const NU se initializeaza in constructor, trebuie sa fie deja initializate, deci se initializeaza in lista constructorului

-obiecte const: nu se schimba
-pentru a se asigura ca starea obiectului nu se schimba functiile de instanta apelabile trebuie definite cu const
-declararea unei functii cu const nu garanteaza ca nu modifica starea obiectului! (se poate modifica prin intermediul pointerului this)

-functii const
-in interiorul lor nu se pot apela functii care nu sunt const

-static
-o singura copie din acea variabila va exista pentru toata clasa
-desi fiecare obiect din clasa poate accesa campul respectiv, in memorie nu avem decat o singura variabila definita astfel
-variabilele initializate cu 0 inainte de crearea primului obiect
-o variabila statica declarata in clasa nu este definita (nu s-a alocat inca spatiu pentru ea), deci trebuie definita in mod global in afara clasei
-aceasta definitie din afara clasei ii aloca spatiu in memorie
-exemplu: class A
{static int a;};
int A::a; //definirea in afara clasei
-pot fi folosite inainte de a avea un obiect din clasa respectiva (folosim operatorul de rezolutie de scop cu numele clasei)

-il facem **static const** si devine similar ca un const la compilare
-static const trebuie initializat la declarare (nu in constructor)

-functii membru static

-au dreptul sa foloseasca doar elemente statice din clasa (sau accesibile global)
-nu putem avea varianta statica si non-statica pentru o functie
-nu au pointerul this, nu sunt asociate unui obiect
-nu pot fi declarate ca "virtual"
-folosire limitata (initializare de date statice)

Tutoriat 1
Noțiuni introductive

1. Cum arată o clasă?

```
class NumeClasă {
        [modificatori de acces]:
        date;
        metode;
} [nume obiecte de tipul
NumeClasă];
```

```
class Person{
private:
   string name;
   double age;
public:
   double getAge(){
      return this->age;
   }
} p1, p2, p3;
```

2. Ce este un obiect?

   Definiție: Un obiect este o **instanță** a unei clase.

   Ce este concret un obiect?
   Un obiect este o *variabilă* de tipul unei clase.

```
int main() {
   Person p1;
   Person *p2 = new Person();
}
```

3. Struct vs class
   a. struct (C)
      i. nu pot conține și metode
      ii. modificator de acces **public** *by default*
      iii. nu permite moștenirea
   b. struct (C++)
      i. modificator de acces **public** *by default*
      ii. permite moștenirea
   c. class (C++)
      i. modificator de acces **private** *by default*
      ii. permite moștenirea

C++

```
struct PersonStruct{
   string name;
   double age;

   string getName() {
      return name;
   }
}ps;

int main() {
   ps.name = "ana";
   ps.age = 18;
   cout<<ps.name<<" "<<ps.age<<endl;
//afiseaza: ana 18
   cout<<ps.getName(); // afiseaza: ana
}
```

4. Principiile POO
   - **Încapsularea** (Encapsulation)
   - Moștenirea (Inheritance)
   - Abstractizarea (Abstraction)

- Polimorfismul (Polymorphism)

5. Încapsularea
   a. Ce este?
      - Toate variabilele și funcțiile sunt înglobate într-o singură structură de date(clasă).
      - Accesul la anumiți membri ai unei clase poate fi controlat(folosim modificatorii de acces)

   b. Cum se face încapsularea?
      - **Modificatorii de acces din C++:**
        - **private** : datele și metodele NU pot fi accesate din afara clasei
        - **protected**: asemănător cu private, dar mai accesibil (to be continued..)
        - **public**: accesul este permis de oriunde
      - **Getters && setters:**
        - **getters** : metode **public** care întorc valoarea unei date membru **private** în afara clasei
        - **setters** : metode **public** care permit modificarea unei date membru **private** din afara clasei

   c. Exemplu de clasă care respectă principiul încapsulării

```
class Person{
   private:
      string name;
      double age;
   public:
      string getName() {
         return name;
      }

      void setName(string name) {
         this -> name = name;
      }

      double getAge() {
         return age;
      }

      void setAge(double age) {
```

```
        this -> age = age;
    }
};
```

PS: Din asta se pică cel mai ușor la colocviu ☹.

6. Constructori

Constructorul este o **metodă** specială **fără tip returnat** (de obicei este **public**), cu sau fără parametrii și poartă numele clasei, care este apelat în momentul creării unui obiect (adică la declarare).

```
class Person {
        ...
  public:
    Person(const string name, double age) {
      this->name = name;
      this->age = age;
    }
        ...
};
```

Constructorul de copiere(CC):

```
ClassName (const ClassName &obj);
```

Când este apelat CC?

- Când un obiect de tipul clasei este returnat prin valoare
- Când un obiect de tipul clasei este dat ca parametru prin valoare unei funcții
- Când un obiect este construit pe baza altui obiect (Person p, b(p))
- Când compilatorul generează un obiect temporal

**Compilatorul de C++ poate face optimizări și nu va apela mereu CC(mai multe despre copy elision, găsiți aici).*

```
class Person{
private:
```

```
    string name;
    int age;
public:
    Person(string name = "ana", int age = 20): name(name), age(age){}
    Person(const Person & ob){
        this -> name = ob.name;
        this -> age = ob.age;
        cout<<"Copy constructor called"<<endl;
    }

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

    void setName(const string &name) {
        Person::name = name;
    }

    int getAge() const {
        return age;
    }

    void setAge(int age) {
        Person::age = age;
    }
};
Person returnPerson(Person ob){ return ob;}

int main()
{
    Person p1; // nu afiseaza nimic
    Person p2(p1); // Copy constructor called
    cout<< p2.getName()<< " "<<p2.getAge()<< endl; // ana 20
    Person p3("ion", 21);  // nu afiseaza nimic
    Person p4(p3); //  Copy constructor called
    returnPerson(p3); // afiseaza de doua ori Copy constructor called
    cout<< p4.getName()<< " "<<p4.getAge()<< endl; // ion 21
    return 0;
}
```

7. Liste de inițializare

Listă de inițializare reprezintă o altă modalitate de a inițializa un câmp de date în constructor. Are și alte funcționalități care vor fi detaliate mai târziu.

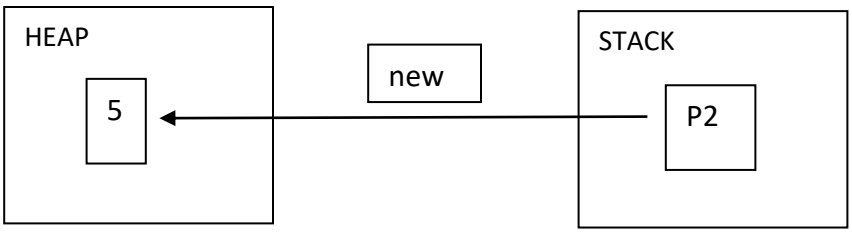
```
class Person {
```

```
        ...
  public:
    Person(const string name, double age) : name(name), age(age) {}
        ...
};
```

8. Pointeri și referințe

Ca să fie mai ușor, pot fi văzute ca tipul de date X* sau X& unde X poate să fie unul dintre tipurile deja definite (int, char, ..) sau un tip nou definit (Person, Animal, Dog..) .

a. Pointeri (*)




```
int main() {
  int *p1= new int;
  int *p2 = new int(5);
  cout<< *p1<< " " <<*p2; // valoarea_aleatoare 5
  delete p2;
  cout<< *p2; // valoare aleatoare
}
```

- Ce este **new**?
  - Keyword care face alocarea dinamică de spațiu pe HEAP.
  - Returnează adresa unde a alocat spațiul în memorie.
- Ce face delete?

  Dezalocă memoria de pe HEAP.

b. Referințe (*)

Extrag si rețin adresa de memorie a unei variabile deja existente.

Nu pot fi considerate variabile noi.

```
int main() {
  int a = 2;
  int &ref = a; // la adresa lui ref pun valoarea a in memorie
  int *p = &a; // in p retin adresa lui a
  int *pp = a; // eroare de compilare
  cout<< *p << endl; // 2
  cout<< ref<< endl; // 2
  cout<< *ref <<endl; // eroare de compilare
}
```

c. Atenție la tipuri

```
int main() {
   int a = 2;
   char c = 'c';
   int * p1 = &a; //2
   char * p2 = &c; // 'c'
   int *p3 = &c; // eroare de compilare
}
```

9. Funcțiile **friend**

Def: Sunt funcții care nu aparțin clasei, definite în afara acesteia, dar care pot accesa membrii privați sau protected ai clasei.

Supraîncarcarea(to be continued...) operatorilor >> și << se face folosind funcții friend, deoarece funcționalitatea acestora este deja definită în biblioteca standard. Pentru a citi sau scrie membrii clasei noastre, numim cele 2 funcții ca fiind friend și le redefinim comportamentul pentru obiectele de tipul clasei noastre.

```
class Person{
   .....
public:
   friend ostream& operator <<(ostream& os, A& ob);//pentru afisare
   friend istream& operator >> (istream& os, A& ob);//pentru citire
   ....
};

istream& operator >> (istream& os, Person& ob)
{
   .....
   return os;
}
ostream& operator <<(ostream& os, Person& ob)
{
   ....
   return os;
}
```

10. **Supraîncarcarea** metodelor în aceeași clasă (Overloading):

Definirea mai multor metode cu același nume în cadrul aceleiași clase. Se face “matching” cu parametrii de la apel(deci aceștia trebuie să difere pentru fiecare metodă în parte). Nu se poate face supraîncarcarea prin metode cu același nume, același tip de parametrii, dar tipuri returnate diferite.

```cpp
class Z{

private:
    int x;
public:
    Z(int x = 2): x(x){}

    void setX(int x){
        this -> x = x;
    }
    void setX(char x){
        this -> x = x;
    }

    int getX() const {
        return x;
    }
};


int main()
{
    Z z;
    z.setX(10);
    cout<<z.getX()<<endl; //10
    z.setX('a');
    cout<<z.getX()<<endl; //97
    return 0;
}
```

Tutoriat 1
Supraîncarcarea operatorilor

1. Operatori unari: -, ++, --, !

```
class A {
private:
    int x;
    bool y;
public:
    A(int x = 2, bool y = true): x(x), y(y){}

    int getX() const {
        return x;
    }

    void setX(int x) {
        A::x = x;
    }

    bool isY() const {
        return y;
    }

    void setY(bool y) {
        A::y = y;
    }

    A operator - (){
        this -> x = - x;
        return A(this -> x);
    }

    A operator ! (){
        this -> y =  !y;
        return A(this -> y);
    }

    A operator ++ (){
        // prefix
        ++ this -> x;
        return A(this -> x, this -> y);
    }

    A operator ++ (int){
        // sufix
```

```
        A copy(this -> x, this -> y);
        this -> x ++;
        return copy;
    }

};
int main()
{
    A a(3, false);
    cout<<a.getX()<<" "<<a.isY()<<endl; // 3 0
    -a;
    cout<< a.getX()<<" "<<a.isY()<<endl; // -3 0
    !a;
    cout<< a.getX()<<" "<<a.isY()<<endl; // -3 1
    ++a;
    cout<< a.getX()<<" "<<a.isY()<<endl; // -2 1
    a++;
    cout<< a.getX()<<" "<<a.isY()<<endl; // -1 1
    return 0;
}
```

2. Operatori binari: +, -, /, *

```
class A {
private:
    int x;
public:
    A(int x = 2): x(x){}

    int getX() const {
        return x;
    }

    void setX(int x) {
        A::x = x;
    }

    // two objects of type A: this and ob
    A operator + (const A& ob) {
        A a;
        a.x = this -> x + ob.x;
        return a;
    }

};
int main()
{
    A a(3), b(5);
    cout<<a.getX()<<endl; // 3
```

```
    cout<<b.getX()<<endl; // 5
    A sum = a + b;
   cout<< sum.getX(); // 8
    return 0;
}
```

3. Operatori relationali: <, > , <=, >=, ==

```
class A {
private:
    int x;
public:
    A(int x = 2): x(x){}

    int getX() const {
        return x;
    }

    void setX(int x) {
        A::x = x;
    }

    // overloaded < operator
    bool operator <(const A& ob) {
        if(this -> x < ob.x){
            return true;
        }

        return false;
    }

};
int main()
{
    A a(3), b(5);
    cout<<a.getX()<<endl; // 3
    cout<<b.getX()<<endl; // 5

   cout<<  (a < b)<<endl; // 1
    //cout<<  (a > b); // eroare (trebuie supraincarcat si > )
    cout<<  (b < a); // 0
    return 0;
}
```

4. Operatorul de asignare =

```
class A {
private:
```

```
    int x;
public:
    A(int x = 2): x(x){}

    int getX() const {
        return x;
    }

    void setX(int x) {
        A::x = x;
    }

    A& operator = (const A& a){
        this -> x = a.x;
        return *this;
    }

};
int main()
{
    A a(3), b(5);
    cout<<a.getX()<<endl; // 3
    cout<<b.getX()<<endl; // 5

   a = b;
    cout<<a.getX()<<endl; // 5
    cout<<b.getX()<<endl; // 5
    return 0;
}
```

5. Input/Output (vezi Tutoriat 1 -> Noțiuni introductive -> funcții friend)

6. Functia call ()

```
class A {
private:
    int x;
public:
    A(int x = 2): x(x){}

    int getX() const {
        return x;
    }

    void setX(int x) {
        A::x = x;
    }
```

```
    // overload function call ()
    A operator()(int e, int f) {
        cout<< " Call () operator overloading"<<endl;
        A a; // x = 2
        cout<<a.x << " "<< e<< " "<< f<<endl;
        a.x =  a.x + e + f; //operatie random
        return a;
    }

};
int main()
{
    A a(3);
    cout<<a.getX()<<endl;

    A c = a(6,7);
    cout<<c.getX()<<endl; // Call () operator overloading
                          // 2  6 7
                          // 2 + 6 + 7 = 15
    return 0;
}
```

7. Operatorul []

```
const int n = 5;

class A {
private:
    int arr[n];

public:
    A() {
        for(int i = 0; i < n; i++) {
            arr[i] = i;
        }
    }

    int &operator[](int i) {
        if( i > n ) {
            cout << "Index out of bounds" <<endl;
            // return first element.
            return arr[0];
        }

        return arr[i];
    }
};

int main() {
```

```
    A a;

    cout << "Value of A[1] : " << a[1] <<endl;  // 1
    cout << "Value of A[3] : " << a[3]<<endl;  // 3
    cout << "Value of A[6] : " << a[6]<<endl;  // Index out of bounds

    return 0;
}
```

8. Opeartorul cast

```
class A
{
    int x;
public:

    A(int val=0): x(val){}

    // Note that conversion-type-id "int" is the implied return type.
    // Returns by value so "const" is a better fit in this case.
    operator int() const
    {
        return this -> x;
    }
};

int main()
{
    A a(10);
    cout << int(a); // 10
    return 0;
}
```

Tutoriat 2
Moștenire și compunere

1. Compunerea

```
class Student {
   ....
};
class Facultate {
   Student s[100]; // compunere
};
```

   a. Ce este?
   Compunerea reprezintă **declararea unui obiect** de tipul unei clase, ca dată membră a altei clase.

2. Moștenirea

   a. Ce este?
      - Al doilea principiu fundamental în POO.
      - Extinderea unei clase, prin crearea altor clase cu proprietăți comune.
      - Se realizează folosind **simbolul :** pus între numele clasei derivate și modificatorul de acces(opțional) alături de numele clasei de bază.

   b. Clasă de bază vs clasă derivată
      - Clasă de bază – cea **din care** se moștenește
      - Clasă derivată – clasa **care** moștenește
      - Obs: Tot ceea ce este **private** în clasa de **bază** devine ***inacesibil*** în clasa derivată.

```
class Animal {
   // clasa de bază
};
class Dog : Animal {
   // clasa derivată
};
```

   c. Tipuri de moșteniri
      i. Moștenire **private**
         - Totul din clasa de bază devine private în clasa derivată.
         - Datele și metodele care sunt deja private în clasa de bază devin inaccesibile în clasa derivată.
         - Tip de moștenire by default când nu este specificat modificatorul de acces la moștenire.
      ii. Moștenire **protected**
         - Tot ce nu e private în clasa de bază devine protected în clasa derivată.
      iii. Moștenire **public**
         - Aceasta este cea mai des folosită moștenire.
         - Totul rămâne la fel ca în clasa de bază.
         - Datele și metodele private tot inaccesibile rămân.

When the component is declared as:	When the class is inherited as:	The resulting access inside the subclass is:
public	public	Public
protected		protected
private		none
public	protected	protected
protected		protected
private		none
public	private	private
protected		private
private		none

d. Constructori la moștenire

- Prima dată, în constructorul din clasa derivată, este apelat constructorul din clasa de bază.
- Se pot apela anumiți constructori din clasa de bază folosind **lista de inițializare.**
- Obs: Este obligatoriu ca în clasa de bază să existe constructorul fără parametrii(dacă nu apelăm explicit alt constructor).

```
class Animal {
    public:
        Animal() {
            cout << "Animal() ";
        }
        Animal(int x) {
            cout << "Animal(x) " << x << " ";
        }
};
class Dog : public Animal {
    public:
        Dog() : Animal(4) {
            cout << "Dog ";
        }
};
int main() {
    Dog d; // Animal(x) 4 Dog
    return 0;
}
```

e. Destructori la moștenire

Destructorii se apelează în ordinea inversă a constructorilor.

```
class Animal {
    public:
        Animal() {
            cout << "C: animal ";
        }
        ~Animal() {
        cout << "D: animal ";
        }
};
class Dog : public Animal {
    public:
        Dog() {
            cout << "C: dog ";
        }
        ~Dog() {
        cout << "D: dog ";
        }
};
int main() {
    Dog d; //C: animal C: dog D: dog D: animal
    return 0;
}
```

f. Moștenirea multiplă

- O clasă poate moșteni mai multe clase în același timp.
- Obs: la moștenirea multiplă, constructorii sunt apelați **în ordinea în care sunt specificați la moștenire**(indifferent de ordinea din lista de inițialziare).

```
class Animal {
    public:
        Animal() {
            cout << "Animal ";
        }
};
class Reptile {
    public:
        Reptile() {
            cout << "Reptile ";
        }
};
class Snake : public Animal, public Reptile {
    public:
        Snake() {
            cout << "Snake ";
        }
};
```

```
int main() {
    Snake s; // Animal Reptile Snake
}
```

3. Mostenire vs compunere

    a. Compunerea: o clasă **are** un obiect de tipul altei clase.
    Ex: Facultatea are mai mulți studenți/profesori/cursuri…
        Firma are mai mulți ingineri/manageri/directori…
    b. Moștenirea: o clasă **este** de tipul altei clase(are câmpuri comune cu aceasta).
    Ex: Animal este câinele/pisica/pasărea....
        Forma geometrică este pătratul/rombul/dreptunghiul...

Tutoriat 3
Const și static

1. Keyword-ul **const**

   a. Ce este?
   Este un *flag(semnal)* dat către compilator care îi transmite să nu permită nicio modificare a valorii unei variabile. Orice **încercare de modificare** a unui const produce o eroare de compilare**.**

   b. La ce folosește?
   Utilizat când avem nevoie ca anumite date să nu poată să fie modificate.

   c. Cum îl folosim?
   O variabilă constantă trebuie **mereu inițializată**, altfel avem o eroare de compilare.

```
const int x = 3;
int const x = 3; // echivalente
```

   d. Pointeri constanți
   Exemplu: pointer constant către un întreg (pointerul nu se poate modifica - adresa de memorie către care pointează nu poate fi modificată, dar își poate schimba valoarea din căsuța de memorie.

```
int* const p = nullptr; // vrea sa fie initializat
int x = 3;
p = &x; // eroare
*p = 5; // ok
```

   e. Pointeri către constante
   Exemplu: pointer către un întreg constant(valoarea din căsuța de memorie **nu** se poate modifica, dar valoarea pointerului - adresa către care arată, se poate modifica).

```
const int* p;
int x = 3;
p = &x; // ok
*p = 5; // eroare
```

f. Diferența dintre cele două:

int * **const** p	**const int*** p

Tip: Te uiți ce tip este lângă keyword-ul **const** (int sau *). Dacă este int atunci întregul este constant, dacă este * atunci pointerul este constant.

g. Date membre constante
   i. Asupra unei date membru constante **NU** se poate aplica **operatorul =** (în afara declarării).
   ii. Se poate inițializa **la declarare** sau prin intermediul **listelor de inițializare**(listele de inițializare sunt *singurele modalități* prin care îi poate fi modificată, ulterior, valoarea unei date membru constante).

```
class A{
    const int x = 2;
    const int y;
public:
    A(int y): y(y){}
    A(int x, int y): x(x), y(y){}

    int getX() {
        return x;
    }

    int getY() {
        return y;
    }
};

int main() {
    A ob1(3);
    A ob2 (5, 6);
    cout<<" Pentru ob1: "<< ob1.getX()<< " "<< ob1.getY()<<endl; // 2 3
    cout<<" Pentru ob2: "<< ob2.getX()<< " "<< ob2.getY()<<endl; // 5 6
    return 0;
}
```

h. Metode constante
   O metodă constantă **NU** are voie să schimbe nimic la datele membre ale pointerului **this**. De aceea, cele mai întâlnite metode constante sunt getters.

   O metodă constantă se definește prin keyword-ul **const** pus între) și {.

   Fără cuvântul cheie **const**, compilatorul va considera metoda **neconstantă**, indiferent dacă modifică sau nu pointerul this.

```
class A{
    ....

   int getX() const { // metoda constanta
      return x;
   }

   int getY() const {
      this -> y = y * 2; // eroare
      return y;
   }

   void mesaj() { // metoda neconstanta (chiar daca nu modifica nimic)
       cout << "Acesta este un mesaj";
   }

};
```

La ce folosesc?
Metodele constante se folosesc pentru a lucra cu **obiecte constante**. Un obiect constant **NU** poate apela o **metodă neconstantă**(pentru că nu îi garantează nimic că nu va încerca să-l modifice în vreun fel), dar un **obiect neconstant**, poate să apeleze oricând o **metodă constantă**.

Adică: obiectul constant – doar metode constante
obiectul neconstant – metode constante și neconstante

```
class A{
    .....

    int getX() const { // metoda constanta
       return x;
    }

    int getY() { // metoda neconstanta (care nu modifica totusi this)
       return y;
    }
};

int main() {
    const A ob1(3);
    A ob2 (5, 6);
    cout<< ob1.getY()<<endl; // eroare: ob1 e const getY() nu este
    cout<<ob1.getX() << endl;  // ok: getX() este const
    cout<< ob2.getY()<< endl // ok: ob2 nu este const
```

```
    return 0;
}
```

i. Return

Valorile returnate de funcții/metode sunt văzute de compilator ca fiind **constante** (dacă **nu** sunt marcate cu **&** la tipul returnat).

```
class Test
{
public:
    Test(Test &) {} // ca sa compileze modificam in Test(const Test &){}
    Test() {}
};
Test fun()
{
    cout << "fun() Called\n";
    Test t;
    return t;
}
int main()
{
    Test t1;
    Test t2 = fun(); // eroare de compilare rezultatul lui fun() este considerat const
    return 0;
}
```

j. Referințe constante

Referința prin definiție presupune că obiectul dat ca parametru la o funcție poate fi modificat în interiorul ei, iar modificările se vor cunoaște și după terminarea funcției.

```
class A{
    int x;
public:
    A(int x): x(x){}
    void modify(A & ob){
        this -> x = ob.x;
       // ob.x = ob.x * 2; // ok
    }
    void notModify(const A & ob){
        this -> x = ob.x;
        //ob.x = ob.x * 2; // eroare
    }
};

int main() {
    A a(2),
```

```
    const A b(3);
    a.modify(b); //eroare
    a.notModify(b); //ok
    return 0;
}
```

Obs: Compilatorul **nu** verifică în interiorul metodei/funcției dacă obiectul chiar este modificat sau nu. El caută **cuvântul cheie** care să garanteze că va rămâne constant și dacă nu îl găsește, întoarce o eroare.

2. Keyword-ul **static**

   a. Ce este?
   O variabilă **statică** se comportă ca o variabilă globală a unei funcții.
   Variabila este inițializată doar **o singură dată** la pornirea programului.

```
void f() {
    static int nr = 0;
    nr++;
    cout << nr << '\n';
}
int main() {
    f(); // 1
    f(); // 2
    f(); // 3
// ...
}
```

   b. Date membre statice

      i. Ce sunt?
      Un membru static este primul inițializat într-o clasă și are aceeași valoare pentru orice instanță a clasei(pratic nu aparține de instanță, ci de întreaga clasă).

      ii. Cum le accesăm din afara clasei (dacă nu sunt private sau protected)?
         1. Prin **operatorul de rezoluție ::**
         2. Printr-o instanță a clasei (este posibil să fie accesate prin orice obiect al clasei declarat în exterior, dar nu pot fi accesate prin this) – nu se reocomandă în practică, dar nu este greșit.

      iii. Statice neconstante

         Cum le inițializăm?

În afara declarației clasei:

```
tip_de_date nume_clasă :: nume_variabilă = valoare_inițială;
```

cu precizarea că dacă **valoarea_inițială lipsește, atunci compilatorul va pune valoarea default. (ex: 0 pentru int)*

sau

În constructorul clasei (în corpul constructorului îi poate fi modificată valoarea, dar **nu și în lista de inițializare**).

Obs: Numai dacă încercăm să accesăm în program o variabilă **statică neinițializată**(adică care **nu** are linia de mai sus în program – cu sau fără valoare_inițială), vom primi o eroare de compilare .

```
class A{
public: // exemplu cu scop didactic
    static int x;
};
int A:: x = 3;
int main() {
    A ob;
    cout<< ob.x<<endl; // ok : 3
    cout<< A:: x << endl; // ok : 3
}
```

iv. Statice constante

1. Cum le inițializăm?
    a. Ca pe cele neconstante
    b. Ca pe o dată membră constantă(doar la declarare sau în corpul constructorului, dar **nu** se poate și prin lista de inițializare din constructor).

```
class A{
public: // exemplu cu scop didactic
    const static int x = 3;
    A(int x): x(x) {} // eroare
};
//const int A:: x = 3;  // ok
int main() {
    A ob;
    cout<< ob.x<<endl; // ok : 3
    cout<< A:: x << endl; // ok : 3
}
```

c. Metode statice

Corpul unei metode statice se poate afla atât în clasă cât și în afara ei.

Orice metodă statică are acces doar la **datele și metodele statice** ale clasei(practic **nu** are pointerul **this**). Orice încercare de a accesa pointerul **this**, în metodele statice, va produce o eroare de compilare.

Accesul se face la fel ca la datele membre statice.

```
class A{
        static int x;
public:
    static int getX() {
        return x;
    }
    static void setX(int x);
};
int A :: x = 3;

void A::setX(int x) {
    A::x = x;
}

int main() {
    A ob;
    cout<< ob.getX()<<endl; // ok : 3
     A:: setX(4) ; // ok
}
```

Tutoriat 4
Overloading și overriding

1. **Supraîncarcarea** (overloading)

   a. Cum se face?
   Prin declararea a 2 sau mai multe metode, în interiorul aceleiași clase, care au **nume identic**, dar antet diferit(**tipul parametrilor diferă/numărul parametrilor diferă**).

   Obs:
   - *Tipul retrunat nu contează*
   - *Supraîncarcarea funcțiilor se poate realiza și în afara claselor(independent de acestea).*

```
class A {
public:
    int f() {
        cout<<"Functia int f()"<<endl;
    }
    void f() {
        cout<<"Functia void f()"<<endl;
    }
};
int main() {
    A a;
    a.f(); // eroare
    return 0;
}
```

```
class A {
public:
    int f() {
        cout<<"Functia int f()"<<endl;
    }
    int f(int x) {
        cout<<"Functia int f(int x)"<<endl;
    }

    int f(double x) {
        cout<<"Functia int f(double x)"<<endl;
    }
```

```
};
int main() {
    A a;
    a.f(); // Functia int f()
    a.f(2); //Functia int f(int x)
    a.f(3.4);//Functia int f(double x)
    return 0;
}
```

b. Ascunderea metodelor (Hiding)

i. Ce face?
Anulează procesul de supraîncarcare.

ii. Când se realizează?
Doar la **moștenirea** claselor.

Obs: Folosim operatorul de rezoluție(**::**) alături de numele clasei de **bază,** dacă dorim apelul metodei din clasa de bază, făcut printr-o instanță a clasei derivate.

```
class Baza {
public:
    void f() {
        cout<<"f() din clasa de baza"<<endl;
    }
};
class Derivata : public Baza {
public:
    void f(int x) {
        cout<<"f(int x) din clasa derivata"<<endl;
    }
};
int main() {
    Derivata d;
    d.f();//eroare
    d.Baza:: f(); //ok -> afiseaza: f() din clasa de baza
    d.f(2); // ok -> afiseaza: f(int x) din clasa derivata

    Baza b;
    b.f(); //ok -> afiseaza: f() din clasa de baza

    return 0;
}
```

c. Operatori care **NU** pot fi supraîncărcați
   - Operatorul * (pointer)
   - Operatorul :: (rezoluție)
   - Operatorul .
   - Operatorul ?:
   - Operatorul sizeof
   - Operatorul typeid

2. **Suprascrierea** (overriding)
   a. Când?
   Suprascrierea se realizează la **moștenirea** claselor.

   b. Ce face?
   Metoda suprascrisă este înlocuită complet de cea care suprascrie.

   c. Cum?
   Metodele clasei **au același nume și aceiași parametri**i.

   Obs: La fel ca la supraîncarcare, folosim operatorul de rezoluție(**::**) alături de numele clasei de **bază**, dacă dorim apelul metodei din clasa de bază, făcut printr-o instanță a clasei derivate.

```
class Baza {
public:
   void f() { cout << "Metoda f() din clasa de baza"<<endl; }
};
class Derivata : public Baza {
public:
   void f() { cout << "Metoda f() din clasa derivata"<<endl; }
};
int main() {
   Derivata d;
   d.Baza::f(); // Metoda f() din clasa de baza
   d.f(); // Metoda f() din clasa derivata
   return 0;
}
```

Tutoriat 4
Upcasting

1. Upcasting

a. Cum se face?

Orice referință/pointer către o clasă **derivată** poate fi convertită într-o referință/pointer către clasa de **bază**.

Adică?

Într-o referință/pointer de tipul clasei de bază rețin adresa către un obiect de tipul clasei derivate:

Clasa_de_bază pointer = new Clasa_derivată();* -> **upcasting dinamic** (cel mai recomandat și frecvent folosit)

```
class Shape {};
class Square : public Shape {};
class Circle : public Shape {};

int main() {
   Shape* s1 = new Square();
   Shape* s2 = new Circle();
}
```

b. Condiția pentru ca upcasting-ul să fie posibil:

Moștenirea să fie de tip **public**.

PS: Apare destul de frecvent un exercițiu cu o astfel de *capcană* la examen.

```
class B{};
class D: private B{};
class C: protected B{};
class A: public B{};
int main(){
   B *p1 = new D(); // error: 'B' is an inaccessible base of 'D'
   B *p2 = new C(); // error: 'B' is an inaccessible base of 'C'
   B *p3 = new A(); //ok
}
```

c. Unde se folosește?
Upcasting-ul este folositor când avem de lucrat cu **vectori de obiecte** care pot fi de mai multe tipuri.

Obs: În exemplu se folosește biblioteca *vector* din STL. Mai multe despre STL puteți afla aici.

```
#include<iostream>
#include<vector>
using namespace std;

class Shape {
   friend ostream& operator <<(ostream& os, Shape& ob);
   friend istream& operator >> (istream& os, Shape& ob);
};
istream& operator >> (istream& os, Shape& ob)
{
   cout<<"citeste datele obiectului"<<endl;
   return os;
}
ostream& operator <<(ostream& os, Shape& ob)
{
   cout<<"afiseaza datele obiectului"<<endl;
   return os;
}
class Square : public Shape {

};
class Circle : public Shape {};

int main() {
   int n;
   cout<<"Dati n= ";
   cin >> n;
   vector<Shape*> formeGeometrice; // vectorul din STL
   for (int i = 0; i < n; ++i) {
      char optiune;
      cout << "Patrat sau cerc (P / C): ";
      cin >> optiune;

      if(optiune == 'P'){
            Square* s = new Square();
            cin >> *s;
```

```
            formeGeometrice.push_back(s);
          }
        else {
            Circle* c = new Circle();
            cin >> *c;
            formeGeometrice.push_back(c);
          }
      }
   cout<<"S-a terminat citirea"<<endl;

   for (int i = 0; i < n; ++i) {
     cout<<*formeGeometrice[i]<<endl;
   }
   return 0;
}
```

d. Suprascriere și upcasting

Printr-o referință/pointer de tipul clasei de bază(nu contează daca conține un obiect de tipul clasei derivate --- upcasting sau de tipul clasei de bază) se accesează doar metoda suprascrisă din clasa de bază.

```
class B{
public:
   void print(){cout<<"Print from B::"<<endl;}
};
class D: public B{
public:
   void print(){cout<<"Print from D::"<<endl;}
};

int main(){

   B *upcast = new D();
   B *b = new B();
   D *d = new D();

   upcast->print();// Print from B::
   (*upcast).print(); // Print from B::
   upcast-> D:: print(); //eroare -> 'D' is not a base of 'B'
   b->print(); // Print from B::
   d->print(); // Print from D::
}
```

e. Upcasting static(nerecomandat în practică, dar există)

Clasa_de_bază ob = (Clasa_de_bază) ob_clasa_derivată;

```
class Shape {};
class Square : public Shape {};
class Circle : public Shape {};
int main() {
    Square s;
    Circle c;
    Shape s1 = (Shape) s;
    Shape s2 = (Shape) c;
    return 0;
}
```

Tutoriat 5
Try-catch

1. La ce folosește?

   Tratarea unor erori neprevăzute care apar în timpul execuției unui program. Acestea duc la oprirea neașteptată a acelui program, dacă nu sunt prinse într-un bloc **try-catch**.

   Programul compilează, însă aruncă o eroare la rulare(runtime) și se oprește din execuție la instrucțiunea care o produce.

```cpp
#include<iostream>
using namespace std;

int main(){
    int x = 10, y = 0;
    cout<<x/y;  //eroare la runtime -> impartire la 0
    return 0;
}
```

```cpp
#include<iostream>
using namespace std;

int main(){
    int x = 10, y = 0;
    try {
        if(y ==0) {
            throw runtime_error("0 division");
        }else{
            cout<<x/y;
        }
    }
    catch(...){
        cout<<"A aparut o eroare"<<endl;
    }
    cout<<"A trecut de impartire";
    return 0;
}
```

   Programul va afișa:

   A aparut o eroare
   A trecut de impartire

2. Pașii de execuție pentru un bloc **try-catch**

1. Se execută, pe rând, ce se află în blocul **try**.
2. Dacă se întâlnește o excepție sau este aruncată una(throw) se oprește execuția blocului **try**.
    2.1. Se caută un bloc **catch** care să prindă exact tipul aruncat(NU se pot produce cast-uri implicite ca: int -> double, char -> int etc..) din **try**.
    2.2. Daca acest tip se găsește, se va executa ce se află în interiorul acelui **catch** și se continuă execuția programului după blocul **try-catch**.
    2.3. Altfel, se oprește execuția programului.
3. Dacă nu se întâlnește nicio excepție în **try**, se sare peste blocul **catch.**

Obs: Instrucțiunea *catch(...){…}* prinde orice tip de eroare.

3. Tipuri de date pentru excepții

În C++, poate fi aruncat orice tip de date de la cele deja definite(int, float, string, const char* etc..), la clase definite în program/clase care moștenesc alte tipuri de excepții deja definite:

```
int main(){
    int x = 10, y = 0;
    try {
        if(y ==0) {
            throw "0 division error";
        }else{
            cout<<x/y;
        }
    }
    catch(const char * error){
        cout<<error<<endl;
    }
    return 0;
}
```

Obs: Când aruncați ceva de forma *"String literal"*, tipul de date corespondent este **const char*, NU string.**

throw "String literal" => catch (const char*)
throw string("String literal") => catch (string)

```
class E{
    string message;
public:
```

```
    E(string message): message(message){}

    const string &getMessage() const {
        return message;
    }

};
int main(){
    int x = 10, y = 0;
    try {
        if(y ==0) {
            throw E("0 divison");
        }else{
            cout<<x/y;
        }
    }
    catch(E& e){
        cout<<e.getMessage()<<endl;
    }
    return 0;
}
```

```
class Exception : public runtime_error {
public:
    // Defining constructor of class Exception
    // that passes a string message to the runtime_error class
    Exception()
          : runtime_error("0 division error")
    {
    }
};

int main(){
    int x = 10, y = 0;
    try {
        if(y ==0) {
            throw Exception();
        }else{
            cout<<x/y;
        }
    }
    catch(Exception& e){
        cout<<e.what()<<endl;
    }
    return 0;
}
```

4. Propagarea excepțiilor

Excepțiile se propagă de la o funcție la alta, în funcție de cum sunt apelate acestea.

*Obs: Propagarea prin funcții se oprește la întâlnirea primului bloc **catch** cu tipul potrivit pentru eroarea propagată.*

Programul de mai jos cere introducerea unei valori pentru y până când aceasta este diferită de 0, pentru a efectua împărțirea :

```
class Exception : public runtime_error {
    ....
};

void verify (int y) {
    if (y == 0) {
        throw Exception ();
    }
}
void readY (int& y) {
    cin >> y;
    verify(y); // (apel 2)
}
int main () {
    int x = 10, y;
    while (true) {
        try {
            readY(y); // (apel 1)
            break; // daca NU a aruncat exceptie se excuta break;
        } catch (Exception  e) {
            // a fost aruncata o exceptie, propagate in functii, prinsa in catch
            cout << e.what();
        }
    }
    cout << x / y;
    return 0;
}
```

Tutoriat 5
Virtual, moștenire diamant și downcasting

1. Virtual

a. Ce este?
Virtual este un **keyword** care a apărut pentru a rezolva multe din problemele din C++ legate de moștenire.

b. Când se folosește?
În fața unei metode din clasa de bază(dar și la mostenirea claselor – vezi moștenirea diamant) și înseamnă că dacă acea metoda va fi **suprascrisă(overriding)** într-o clasă derivată, în momentul realizării **upcasting-ului**, metoda apelată va fi **cea din clasa derivată**.

Obs:
- **Virtual** este utilizat, în general, în clasele de bază, pentru că el ajută la moștenire.
- Nu este recomandată folosirea lui într-o clasă care nu urmează să fie moștenită, dar totuși, nu este interzis acest lucru.

```
class A
{
public:
    void f1() { cout << "f1 normal din A"<<endl; }
    virtual void f2() { cout << "f2 virtual din A"<<endl; }
};
class B : public A
{
public:
    void f1() { cout << "f1 din B care suprascrie"<<endl; }
    void f2() { cout << "f2 din B care suprascrie virtual din A"<<endl; }
};
int main()
{
    A *a = new B; // upcasting
    a->f1(); // f1 normal din A
    a->f2(); // f2 din B care suprascrie virtual din A
    return 0;
}
```

c. **Destructor virtual** vs destructor obișnuit(doar la realizarea upcasting-ului)

În mod normal, la moștenire, destructorii sunt *apelați de la clasa derivată spre clasa de bază*. Însă, la **upcasting**, la distrugerea pointerului/referinței prin care se realizează upcating-ul, se va apela **doar** destructorul clasei de bază, cel din clasa derivată rămânând neapelat => *memory leaks*.

Dacă destructorul din clasa de bază este declarat ca fiind și virtual, la distrugerea pointerului/referinței prin care s-a reazlizat upcastingul se vor apela ambii destructori în oridinea corectă, evitând memory leaks.

```
// ------------------- Exemplu normal ------------------
class B {
public:
   ~B() {
      cout <<
          "~B()";
   }
};
class D : public B {
public:
   ~D() {
      cout <<
          "~D()";
   }
};
// ---------------------- Exemplu destructor virtual ------------
class BV {
public:
  virtual ~BV() {
      cout <<
          "~BV()";
   }
};
class DV : public BV {
public:
   ~DV() {
      cout <<
          "~DV()";
   }
};
int main() {
   B *p = new D(); // upcasting
   BV *pv = new DV();
   delete p; // ~B()
   cout<<endl;
   delete pv; // ~DV()~BV()
```

```
    return 0;
}
```

2. Moștenirea diamant

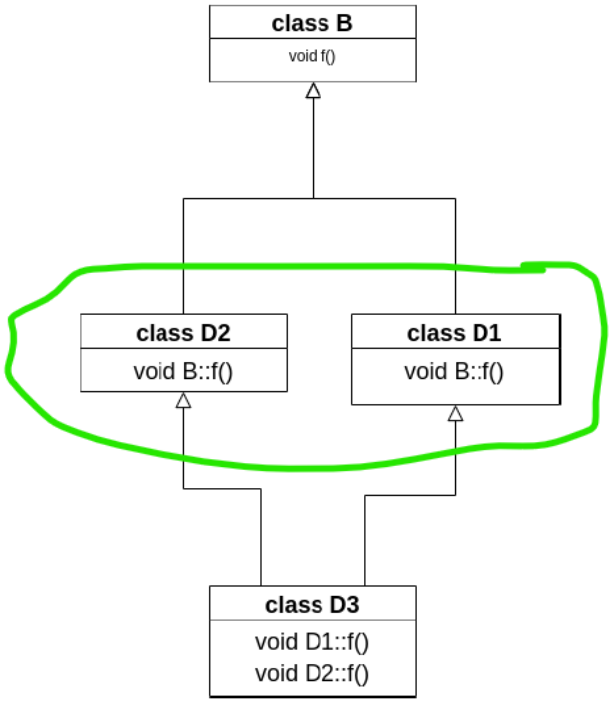
a. Moștenirea virtuală

Este folosită pentru un caz particular de moștenire și anume **moștenirea multiplă**.

Programul de mai jos **nu compliează**, deoarece nu știe care metodă *f()* să apeleze pentru instanța clasei D3(cea venită din D1/ cea venită din D2).

```
class B {
public:
    void f() { cout << "f() din B"; }
};
class D1 : public B {};
class D2 : public B {};
class D3 : public D1, public D2 {};
int main() {
    D3 d3;
    d3.f(); // eroare
    return 0;
}
```

Lanțul de moșteniri pentru programul de mai sus ar arăta așa:



b. Probleme la moștenirea multiplă
În exemplul de mai sus, în momentul moștenirii multiple, clasa D3 va avea acces la metoda *f()*, atât de pe ramura moștenirii clasei D1, cât și pe ramura clasei D2.

c. Soluții pentru rezolvarea problemei diamantului

i. Folosirea operatorului de rezoluție(::) împreună cu numele clasei din care folosim metoda f() moștenită, adăugate la apelul metodei (nu respectă principiile POO, deși este posibilă, **nu** este recomandată).

ii. Folosirea **moștenirii virtuale** (recomandată) : keyword-ul **virtual**(adăugat la moștenirea claselor încercuite cu verde în figura de mai sus) asigură faptul că nu se va copia decât o singură dată metoda *f()* din clasa de bază.

```
class B {
public:
    void f() { cout << "f() din B"; }
};
class D1 : virtual public B {}; //aici
class D2 : virtual public B {}; //aici
class D3 : public D1, public D2 {};
int main() {
    D3 d3;
    d3.f();  // f() din B
    return 0;
}
```

3. Downcasting

a. Ce este?
Reprezintă trecerea de la un pointer/referință de tipul clasei de bază la unul de tipul clasei derivate.

Adică?

Transform un obiect de tipul clasei de bază într-un obiect de tipul clasei derivate.

b. Cum se realizează?
   - Prin intermediul operatorului **dynamic_cast**:

clasa_derivată * pointer = dynamic_cast<clasa_derivată *>(obiect_clasă_de_bază)

!Obs:
- *Instrucțiunea returnează NULL dacă nu se poate face conversia cu succes.*
- *Este necesar ca în clasa de bază sa existe **cel puțin o metodă virtuală**, altfel utilizarea operatorului va duce la o **eroare de compilare**.*
- *Nu se poate realiza downcastingul(compilează, dar valoarea este NULL) dacă pointerul/referința prin care se face nu reprezintă un upcasting.*

```
class Animal {
public:
    void sleep() { cout << "Sleep"<<endl; }
    virtual ~Animal() {}
};
class Dog : public Animal {
public:
    void bark() { cout << "Bark"<<endl; }
};
class Cat : public Animal {
public:
    void meow() { cout << "Meow"<<endl; }
};
int main() {
    Animal* animals[2];
    Animal* a = new Animal();
    cout<< "Downcasting fara upcasting: "<<
dynamic_cast<Cat*>(a)<<endl; // afiseaza valoarea 0 (adica NULL)

    animals[0] = new Cat();
    animals[1] = new Dog();

    for (int i = 0; i < 2; i++) {
        if (Dog* d = dynamic_cast<Dog*>(animals[i])) {
            d->bark();
        } else if (Cat* c = dynamic_cast<Cat*>(animals[i])) {
            c->meow();
        }
    }
```

```
    return 0;
}
```

- Același lucru se poate realiza și prin intermediul **operatorului static_cast<>()**, însă este folosit doar în anumite situații. (mai multe puteți citi aici).

- Downcasting fără metode virtuale și fără operatorul dynamic_cast

  Este posibil doar în cazurile în care știm sigur ce tip de obiect se află pe fiecare poziție a vectorului.

```
class Animal {
public:
    void sleep() { cout << "Sleep"<<endl; }
};
class Dog : public Animal {
public:
    void bark() { cout << "Bark"<<endl; }
};
class Cat : public Animal {
public:
    void meow() { cout << "Meow"<<endl; }
};
int main() {
    Animal* animals[2];
    animals[0] = new Dog();
    animals[1] = new Cat();
    for (int i = 0; i < 2; ++i) {
        if (i == 0) {
            Dog* d = (Dog*)animals[i]; // downcast
            d->bark(); // correct
        } else {
            Cat* c = (Cat*)animals[i]; // downcast
            c->meow(); // correct
        }
    }
    return 0;
}
```

Tutoriat 6
Polimorfism

1. Ce este?
   Abilitatea unui obiect de a lua mai multe forme.

2. Cum se realizează?
   - **supraîncarcare/suprascriere** de metode
   - **supraîncarcare** de operatori
   - metode **virtuale**
   - **template**

3. Tipuri
   - Polimorfism la compilare (compile time polymorphism/**late binding**)
     Se realizează prin 2 modalități:
     - ✓ **supraîncarcare** de metode
     - ✓ **supraîncarcare** de operatori
     - ✓ **template**

     Obs:
     - ❖ Supraîncarcarea se poate realiza atât în clasă, cât și în afara acesteia.
     - ❖ Vezi operatori care nu pot fi supraîncarcați în Tutoriat 4- *Suprascriere și supraîncarcare.pdf*
     - ❖ Vezi supraîncarcarea operatorilor în Tutoriat 1 - *Supraîncarcarea operatorilor.pdf*

   - Polimorfism la rulare (run time polymorphism/**early binding**)
     Se realizează prin 2 modalități:
     - ✓ **suprascriere** de metode
     - ✓ metode **virtuale**

4. **RTTI**(Run Time Type Information)
   a. Ce este?
      Un mecanism care expune informații despre tipul unui obiect la execuție.

   b. Când se folosește?
      Este disponibil doar pentru clasele care conțin **cel puțin o metodă virtuală**.

   c. Exemplu

dynamic_cast: Deduce tipul real de la upcasting(dreapta egalului) al obiectului la execuție(are nevoie de cel puțin o metodă virtuală în clasa de bază).

Tutoriat 6
Abstractizare

1. Ce este?
   Ascunderea detaliilor implementării față de utilizatorul unei clase.

2. Exemple din practică
   Ex1: Când includem o bibliotecă, folosim funcțiile/metodele din acea bibliotecă, fără a fi nevoie să cunoaștem implementarea acestora.

   Ex2: Când scriem o clasă într-un fișier de tip header(.h), în programul principal doar apelăm metodele publice, fără a fi nevoie să cunoaștem implementarea acestora.

3. Metode pur virtuale
   Sunt metode virtuale fără implementare.

```
virtual void nume_metodă() = 0;
```

4. Clase abstracte
   Clase care conțin **cel puțin o metodă pur virtuală**.

   Obs:
   - **Nu** pot fi instanțiate.
   - Clasele lor derivate pot fi instanțiate doar dacă au fost **implementate toate metodele pur virtuale.**

5. Clasă abstractă vs interfață(în C++):
   Interfața are **doar metode pur virtuale**, spre deosebire de clasele abstracte care pot avea și alte tipuri de metode, pe lângă cele pur virtuale.

```
class MyInterface
{
public:
    // Empty virtual destructor for proper cleanup
    virtual ~MyInterface() {}

    virtual void Method1() = 0;
    virtual void Method2() = 0;
};


class MyAbstractClass
```

```
{
public:
    virtual ~MyAbstractClass();

    virtual void Method1();
    virtual void Method2();
    void Method3();

    virtual void Method4() = 0; // make MyAbstractClass not instantiable
};
```

6. Exemplu abstractizare

```
class Animal {
public:
    virtual void eat() = 0;
    virtual void sleep() = 0;
};
class Dog : public Animal {
public:
    void eat() {
        cout << "Dog::eat()";
    }
    void sleep() {
        cout << "Dog::sleep()";
    }
    void bark() {
        cout << "Dog::bark()";
    }
};
class Cat : public Animal {
public:
    void eat() {
        cout << "Cat::eat()";
    }
    void sleep() {
        cout << "Cat::sleep()";
    }
    void meow() {
        cout << "Cat::meow()";
    }
};
int main() {
    Animal a; // eroare de compilare => Animal este clasa abstracta=> nu poate fi instantiata
    Dog d; // corect
    Cat c; // corect
    Animal* a1 = new Dog; // corect (se declara un pointer de tipul clasei abstracte, dar se intasntiaza un obiect in memorie de tipul clasei neabstracte)
```

```
    Animal* a2 = new Cat; // corect
    return 0;
}
```

Tutoriat 6
Template

1. Ce este?
   Presupune scrierea unei singure clase/funcții a cărei comportament este asemănător și se modifică, doar dacă se modifică și un anumit tip de date.

   Obs:
   Template este o formă de **polimorfism** la compilare.

2. La ce se folosește?
   Template este un instrument prin care se poate evita rescrierea unor blocuri de cod.

3. Despre **typename**
   Poate fi înlocuit începând cu standardul C++17 cu **class(cel folosit în definirea template)** fără vreo diferență semnificativă.

4. Funcții template
   a. Pași la compilare:

      1. Când e întâlnită o funcție template, este compilată, fără a se ține cont de tipul de date necunoscut.
      2. În momentul în care se apelează o funcție template, compilatorul creează o nouă funcție obișnuită în care tipul de date necunoscut este înlocuit cu cel specificat în apel.

      f<int>(3) ===> void f (int x) {...}
      f<char>('c') ===> void f (char x) {...}

```
template <typename T>
void f (T x) {
    cout << "Functie template"<<endl;
}
void g (int x) {
    cout << "Functie obisnuita"<<endl;
}
int main () {
    f<int>(3); // Functie template
    f<char>('c'); // Functie template
    g(2); // Functie obisnuita
```

```
    return 0;
}
```

b. Specializarea funcțiilor template

```
template <typename T>
void f (T x) {
    cout << "Functie template"<<endl;
}
template <>
void f (int x) {
    cout << "Functie  specializata"<<endl;
}
int main () {
    f(3); // Functie  specializata
    f<int>(3); // Functie  specializata
    f('c'); // Functie template
}
```

c. Prioritatea la supraîncarcare(overloading)

Cum procedează compilatorul când caută o funcție care se potrivește cu un apel:

1. Se caută o **funcție normală** care să aibă parametrii potriviți.
2. Dacă nu s-a găsit la punctul 1, se caută o **specializare** cu parametrii potriviți.
3. Dacă nici punctul 2 nu a furnizat un rezultat, se caută **o funcție template** cu numărul de parametrii potriviți.
4. Dacă nici punctul 3 nu a furnizat un rezultat, se întoarce o **eroare la compilare.**

Mai pe scurt, o ordine de prioritate ar fi:

1. Funcțiile normale
2. Funcțiile specializate
3. Funcțiile template

```
template <typename T>
void f (T x) {
    cout << "Functie template"<<endl;
}
template <>
void f (int x) {
    cout << "Functie specializata"<<endl;
}
void f (char c) {
```

```
    cout << "Functie normala (char)"<<endl;
}
void f (int c) {
    cout << "Functie normala (int)"<<endl;
}
int main () {
    f(3); // Functie normala (int)
    f('c'); // Functie normala (char)
    f<int>(3); // Functie specializata
    f<char>('c'); // Functie template
}
```

5. Clase template
    a. Exemplu

Obs:
Spre deosebire de funcțiile template, aici este obligatorie specificarea tipului de date la declararea unui obiect (între <>).

```
template <typename T>
class A {
// clasa template
    T x;
    int y;
public:
    A(){cout<<"A"<<endl;}
};
class B {
// clasa obisnuita fara template
    char x;
    int y;
public:
    B(){cout<<"B"<<endl;}
};
int main () {
    A<int> a1; // A
    A<char> a2; // A
    B b; // B
    return 0;
}
```

    b. Specializarea claselor template

```
template <typename T>
class A {
    T x;
```

```
public:
    A() { cout << "A template"; }
};
template <>
class A<int> {
    int x;
public:
    A<int>() { cout << "A specializata "; }
};
int main () {
    A<int> a1; //A specializata
    A<char> a2; // A template
}
```

c. Metode template

```
template <typename T>
class A {
    T x;
public:
    T getX () const;
    void setX (T);
};
template <typename T>
T A<T>::getX () const {
    return x;
}
template <typename T>
void A<T>::setX (T _x) {
    x = _x;
}
int main(){
    A<int> object;
    object.setX(2);
    cout<<object.getX(); // 2
}
```

Tutoriat 6
Singleton

1. **Design patterns**
   Sunt soluții tipice pentru probleme comune în dezvoltarea de software. Acestea sunt independente de limbaj, pentru că se referă mai mult la proiectarea unor clase, decât la modul în care acestea sunt implementate. Adică, un design pattern din C++ este același și în Java, cu diferențe minore de implementare.

   Tipuri:
   - Creational Patterns
   - Structural Patterns
   - Behavioral Patterns

2. **Creational Patterns**
   Oferă diverse mecanisme de creare a obiectelor. Astfel, este mai ușoară întreținerea codului care poate fi refolosit mai târziu.

3. **Singleton**
   a. Ce este?
      - design pattern creațional
      - permite crearea unei clase care are în *permanență* o **singură instanță**

   b. La ce folosește în practică?
      - menține o singură conexiune activă la o bază de date
      - menține un singur flux deschis la un fișier comun mai multor procese
      - menținerea unei singure instanțe pentru **meniul** unui joc/unei aplicații

   c. Cum se implementează?
      1. Se creează un **constructor privat** pentru a împiedica crearea de noi instanțe prin keyword-ul *new.*
      2. Se creează un câmp de date **static** care va fi un **pointe**r către un obiect de **tipul clasei** și care va reprezenta instanța unică a acelei clase.
      3. Se creează o metoda **statică** care apelează **constructorul privat** dacă nu a fost creată deja o instanță a clasei.

```
class Singleton {
private:
   static Singleton* instance;
   Singleton() {
      cout << "Constructor called";
   }
public:
   static Singleton* getInstance() {
      if (instance == NULL) {
         instance = new Singleton;
      }
      return instance;
   }
};
Singleton* Singleton:: instance;

int main () {
   Singleton *s1;
   s1 = Singleton::getInstance(); // Constructor called
   Singleton *s2; //pointer, nu are o zona de memorie catre care arata la
linia asta
   s2 = Singleton::getInstance(); // nu intra in constructor
//    Singleton s3; // constructorul e privat => eroare
//    Singleton *s4 = new Singleton();  // constructorul e privat => eroare
}
```

d. Particularizare
O clasă care permite crearea a maxim 3 instanțe.

```
class Singleton3{
private:

   static Singleton3* _instance[3]; // declaring 3 pointers to Singleton3
   static int count; // the current number of instances created

   Singleton3(){cout<<"Constructor called"<<endl;}

public:
   static Singleton3* getInstance(){
      if (count < 3){
         _instance[count] = (Singleton3*) new Singleton3();
         count++;
         return (Singleton3*)(_instance[count - 1]);
      }
      else {
         return nullptr;
      }
```

```
    }

    ~Singleton3(){
        delete  [] *_instance;
    }
};
Singleton3* Singleton3 :: _instance[] = { nullptr,nullptr,nullptr };
int Singleton3 :: count = 0;

int main() {
    Singleton3* objArray[3];

    for (int i = 0; i < 3; i++){
        // Create an instance
        objArray[i] = Singleton3::getInstance();
    }

    // Attempt to create object no 4
    Singleton3* obj4 = Singleton3::getInstance();
    if (obj4 == nullptr) {
        cout << "Error of creating obj4." << endl;
    }

    return 0;
}
```

Tutoriat 8

Cheat sheet

Acest material reprezintă chestiuni de teorie de care aveți nevoie în rezolvarea exercițiilor de la examen

Acestea NU sunt copiuțe pentru examen!

1. **Noțiuni introductive**
   Struct (C++) – accesul default este **public**
   Class (C++) – accesul default este **private**

   Funcțiile friend pot accesa și membrii private sau protected ai unei clase.

2. **Moștenire**
   - ✓ Presupune clasa de bază și clase derivate din aceasta. (una sau mai multe)
   - ✓ **Private** în clasa de bază => **inaccesibil** în clasa derivată.
   - ✓ **Protected** în clasa de bază => **accesibil** în clasa derivată.
   - ✓ Tipurile de moștenire(public, private, protected) influențează accesul datelor și metodelor moștenite în clasa derivată.
   - ✓ Constructorii sunt apelați din clasa de bază spre derivată. Destructorii sunt apelați invers constructorilor.

3. **Compunere**
   Instanțiere de obiecte de tipul unei clase în altă clasă.

4. **Keyword-ul const**
   - ○ Orice încercare de modificare a unui obiect constant => eroare de compilare.
   - ○ Variabilele constante trebuie să fie mereu inițializate.
   - ○ Pointer constant către un tip oarecare (adresa nu se modifică):
     *tip * const nume_pointer;*
   - ○ Pointer către un tip oarecare constant (zona de memorie nu se modifică):
     *const tip * nume_pointer;*
   - ○ Datele membre constante pot fi inițializate/modifică valoarea doar la declarare sau prin lista de inițializare.

- Metodele constante nu au voie să schimbe nimic la datele pointerului *this*.

5. **Keyword-ul static**
   - ✓ Variabila statică este inițializată o singură dată la pornirea programului.
   - ✓ Are nevoie de linia de inițializare din afara clasei, altfel => eroare de compilare.
   - ✓ Pot fi accesate prin instanțe sau numele clasei cu ::
   - ✓ Aceeași valoare pentru toate instanțele clasei.
   - ✓ Metodele statice nu au *this* => lucrează doar cu date și metode statice.

6. **Supraîncărcarea (overloading)**
   - Metode cu același nume dar parametrii diferiți (nu contează tipul returnat).
   - Operatorii **, ::, . , ?:, sizeof, typeid* **nu** pot fi supraîncărcați.
   - **Hiding**
     - Supraîncărcare la moștenire.
     - Printr-o instanță de tipul clasei derivate accesăm metoda supraîncărcată din clasa derivată.
     - Printr-o instanță de tipul clasei derivate folosim numele clasei de bază și *operatorul ::* pentru a accesa metoda supraîncărcată din clasa de bază.

7. **Suprascriere (overriding)**
   - ✓ Se realizează numai la moștenirea claselor.
   - ✓ Metode cu același nume și aceiași parametrii.
   - ✓ Printr-o instanță de tipul clasei derivate accesăm metoda suprascrisă din clasa derivată.
   - ✓ Printr-o instanță de tipul clasei derivate folosim numele clasei de bază și *operatorul ::* pentru a accesa metoda suprascrisă din clasa de bază.

8. **Upcasting**
   - ✓ Upcasting dinamic:
     clasă_de_bază pointer = new clasă_derivată();*
   - ✓ Moștenirea trebuie să fie de tip public, altfel => eroare de compilare.

- ✓ Suprascriere/supraîncărcare + upcasting => este accesibilă doar metoda din clasa de bază.
- ✓ Upcasting static (nerecomandat):
  clasă_de_bază ob = (clasă_de_bază) ob_clasă_derivată;

9. Metode virtuale

- ❖ Se folosesc în clasele de bază cu metode care urmează a fi suprascrise în clasele derivate.
- ❖ Nu produce erori dacă este folosit și la metode din clase care nu urmează a fi moștenite.
- ❖ Virtual + suprascriere + upcasting => apel metodă din clasa derivată.

10. Moștenirea diamant

Keyword-ul **virtual** la moștenirile de la mijlocul diamantului rezolvă problema => nu se mai moștenesc duplicate în clasa cea mai de jos din ierarhie

11. Downcasting

*clasă_derivată * pointer=dynamic_cast<clasă_derivată*>(obiect_clasă_de_bază);*

- Operatorul **dynamic_cast** retrunează NULL dacă nu se poate face conversia.
- În clasa de bază trebuie să existe **cel puțin** o metodă/ constructor/ destructor virtual, altfel => eroare de compilare.
- Pointerul pe care se face conversia trebuie să reprezinte un upcasting, altfel **dynamic_cast** retrunează NULL
- Downcasting fără operatorul **dynamic_cast**, fără metode virtuale și fără upcasting – este posibil sub forma:
  *clasă_derivată * pointer = (clasă_derivată*) pointer_clasă_de_bază;*

12. Abstractizare

- ❖ Clase care au cel puțin o metodă pur virtuală:
  virtual void nume_metodă() = 0;
- ❖ **NU** pot fi instanțiate.
- ❖ Clasele care le moștenesc trebuie să implementeze toate metodele pur virtuale, altfel nici ele nu pot fi instanțiate.

13. Template

- ❖ Typename este același lucru cu class (standarde C++ diferite).

- La apelul de funcții nu este obligatorie specificarea *<tip>*, dar la clase este.
- Ordinea de potrivire la compilare:
    1. Funcții normale
    2. Funcții specializate
    3. Funcții template

TEORIE POO(toate variantele din 2004-prezent)

Subiect 2004:

1) Prin ce se caracterizeaza o variabila statica a unei clase:

Proprietetati:

- variabilele statice sunt precedate de cuvantul- cheie static.
- o singura copie din acea variabila va exista pentru toata clasa
- o variabila statica declarata in clasa nu este definita(nu are zona de memorie alocata). Pentru a putea fi definite, sunt redeclarate in exteriorul clasei respective, folosind operatorul de rezolutie.
- Dat fiind ca variabilele statice sunt instantiate indiferent daca sunt declarate sau nu obiecte, putem accesa variabilele statice fara a ne folosi de vreun obiect, prin intermediul operatorului de rezolutie.

Exemplu:

```
  Class A
{
   static int x;
}
int A::x=5;
```

Exemplu de utilizare:

pentru a retine numar de instante ale unui obiect.(si a limita instatierea mai multor obiecte ex Singletone)

2)Prin ce se caracterizeaza o metoda statica a unei clase?

Sintaxa:

```
class A{

static void numara();

}
```

Proprietati:

-functiile statice pot opera doar asupra variabilelor statice din clasa.

- in cadrul acestor functii nu exista pointerul *this , pentru ca aceste metode se apeleaza indiferent daca exista sa nu obiecte instantiate.

-functiile statice nu pot avea natura virtuala

- nu pot exista mai multe variante ale aceleasi functii, una statica si una nestatica.

Exemplu de utilizare:

metodele statice sunt utilizate pentru accesarea datelor statice, care au fost declarate ca fiind private(principiul incapsularii)

3) Descrieti pe scurt diferenta dintre transferul prin valoare si transferul prin referinta in cazul apelului unei functii.

TRANSFER PRIN VALOARE	TRANSFER PRIN REFERINTA
-copierea valorii transmise ca parametru actual in parametrul formal(care este creat pe stiva la lansarea executiei functiei), ca o variabile locala. -pot aparea expresii sau nume de variabile.	-se evita consumul de resurse necesar pentru creerea unei alta variabile. - variabila transmisa prin referinta constanta nu va fi modificata -**NU** pot aparea expresii, ci doar nume de variabile(avem eroare logica)
sintaxa transmitere prin varloare: *int suma(int a, int b){..}*	**sintaxa transmitere prin referinta:** *void suma(int &S ,int a,int b){..}*

OBS: Daca transmitem obiecte prin apel prin referinta, nu se mai creeaza noi obiecte temporare, ci se lucreaza direct pe obiect trimis ca referinta.(nu se mai apeleaza copy-constructorul si nici destructorul).

4) Spuneti care este diferenta dintre incluziune de clase si mostenire de clase si cand se foloseste fiecare metoda.

INCLUZIUNEA DE CLASE	MOSTENIREA DE CLASE
- in cadrul unei clase avem campuri de date de tipul alor clase. -cand vrem ca doar anumite parti ale unei clase sa fie incluse in declararea unei clase noi	-mecanismul prin care o clasa este creata prin preluarea tuturor elementelor unei alte clase si adaugarea unor elemente noi. -clasa de la care se pleaca se numeste clasa de baza, iar clasa la care se ajunge este clasa derivata. la derivare putem asocia clasei un **atribut de acces.(vezi tabel)**
sintaxa: *class A{...};* *class B* *{* *A o1;* *int x;* *}*	**sintaxa** *class A{...};* *class B: public A* *{...}*

Accesul asupra membrilor mosteniti:

When the component is declared as:	When the class is inherited as:	The resulting access inside the subclass is:
public	public	Public
protected		protected
private		none
public	protected	protected
protected		protected
private		none
public	private	private
protected		private
private		none

5) Spuneti care dintre urmatoarele reprezinta mecanisme prin care se obtine polimorfismul de functii:

functiile friend , functiile inline , constructorii, functiile virtuale, destructorii.

Raspuns: functii inline,constructori, functii virtuale.

Obs: nu poate exista decat un singur destructor pentru fiecare clasa.

Subiect 2005(Mate info)

6) Descrieţi pe scurt diferenţa dintre funcţiile care returnează valoare şi cele carereturnează referinţă.

INTOARCERE PRIN VALOARE	**INTOARCERE PRIN REFERINTA**
-copierea valorii furnizate de functie intr-o variabila temporara,de tipul funtiei.	- crearea unei referinte temporare de tipul functiei in functia apelanta , catre variabila intoarsa ca rezultat de functia apelata.

7) Descrieţi pe scurt cum se pot defini funcţii de conversie între tipuri (clase).

Conversiile de tip se pot realiza prin:

- supraincarcarea operatorului unar „cast”(se poate face conversia dintr-un tip clasa intr-un tip fundamental sau intr-un alt tip clasa)
  sintaxa:
  operator TipData();
- prin intermediul constructorilor (consta in definirea unui constructor care primeste ca parametru tipul la care se face conversia). Constructorul intoarce intotdeauna un rezultat de tipul clasei de care apartine, prin urmare prin intermediul cosntructorilor se poate face conversia doar de la un tip fundamental sau un tip clasa la un alt tip clasa.
  OBS: in cazul conversiei de la un tip clasa la alt tip clasa, constructorul trebuie sa aiba acces la campurile private ale tipului clasa in care de

doreste a se face conversia si prin urmare trebuie declarat ca functie friend.

Exemplu :

Fie clasa B (tip de date predefinit sau clasa) si se doreste a se face conversia la tipul A

1) supraincarcam operatorul de cast al tipului in clasa B.

```
class B
{
        public : operator A();
}
```

2) supraincarcare constructorului de copiere cu un singur parametru de tip B:

```
A(B);
```

OBS: Constructorii cu un parametru sunt tratati ca metode de conversie implicita, chiar daca nu pentru asta au fost creati. Solutie: se pune in fata constructorului cuvantul explicit.

8)Spuneti care este diferenta dintre o clasa generica(template) si o calsa abstracta si in ce situatie se foloseste fiecare dintre ele:

CLASA GENERICA(TEMPLATE)	**CLASA ABSTRACTA**
-o clasa templete este un sablon utilizat pentru generarea unor clase concrete, care difera prin tipul anumitor date membre -este o metoda de implementare a polimorfismului de compilare	-clasa abstracta este o clasa in care avem cel putin o metoda virtuala pura. -reprezinta un tip incomplet care este folosit ca fundament pentru calsa derivata. - este o metoda a polimorfismului de executie
Sintaxa: *template<class Tip>class nume-clasa { ....}*	**Sintaxa:** *class A{ virtual f()=0}*

Odata ce am definit o clasa template, putem crea un exemplar al acesteia folosind: nume-clasa<tip obiect>ob;	Clasele virtuale sunt folosite cand cand ceea ce dorim sa modelam are caracteristici generale.

SUBIECT 2007

8) Spuneţi ce reprezintă o funcţie virtuală şi în ce condiţii o funcţie virtuală defineşte o clasa abstractă.

O functie virtuala este o functie care este definita in clasa de baza folosind cuvantul-cheie „virtual" si redefinita in clasa derivata. Redefinirea functiei in clasa derivata are prioritate in fata definitiei din clasa de baza.

- functiile virtuale reprezinta o modalitate de implentare a polimorfismului de executie.
- O functie virtuala pura este o functie care nu are definitie in calsa de baza.

<u>Sintaxa pentru functie virtuala pura:</u>

virtual nume_functie()=0;

-functiile virtuale pure trebuie redefinite in clasa derivata.

- o clasa care contine cel putin o functie virtuala pura este o clasa abstracta(si nu poate fi instantiata)

9) Spuneţi ce reprezintă o funcţie prietenă (friend) a unei clase.

- o functie friend are acces la membrii private si protected ai unei clase, fara sa fie funtie membra a clasei respective.

- pentru a declara o functie friend includem prototipul ei in clasa respectiva, precedat de cuvnatul cheie „friend".

ex: supraincarcarea operatorului „>>" pentru citire, respectiv „<< „ pentru afisare.

```
friend ostream &operator<<(ostream&os, const clasa& p);
```

10) Spuneti daca o variabila constanta poate fi transmisa ca parametru al unei functii si daca da, in ce situatie. Justificati !

O variabila constanta poate fi transmisa ca parametru al unei functii doar cand tipul parametrului functiei :

- **NU** este de tip referinta neconstanta (avem eroare pentru ca incercam sa convertim o variabila constanta la o referinta neconstanta)
- **NU** este de tip neconstant(se apeleaza constructorul de copiere care este de tip A(A &a) , prin urmare avem aceeasi eroare).

11) Descrieti pe scurt ce reprezinta obiectul implicit al unei metode:

- obiectul implicit este obiectul care apeleaza metoda.
- in cadrul metodei nu mai trebuie specificat carui obiect apartin campurile, pentru ca se subintelege ca ele apartin cuvantului implicit.
- obiectul implicit al unei metode se poate apela si folosind pointerul *this.
- este transmit in metoda prin referinta(referinta constanta daca metoda este constanta)

Exemplu:
pentru variabile : this->a;
pentru metode : this->f();

12) Descrieţi pe scurt ce reguli verifică supraîncărcarea operatorilor.

- nu putem defini operatori noi.(doar supraincarca pe cei existenti)
- nu putem modifica numarul operanzilor unui operator existent si nici sintaxa lor.
- nu putem modifica prioritatea si nici asociativitatea operatorilor existenti.

-exista operatori care **NU** se pot supraincarca:

exemplu : „ .” „ :: ” „ , ” „ ?: ”

-operatorii pot fi supraincarcati ca metode sau ca functii indepenente.

OBS: Nu toti operatorii care pot fi supraincarcati ca metode pot fi supraincarcati ca functii independente:

exemplu : „ =” „()” „[]” „->”

SUBIECT 2008-CTI

13) Descrieti trei metode de proiectare diferite prin care elementele unei unei clase se pot regasi in dublu exemplar, sub diferse forme, in definitia altei clase.

A. mostenire multipla, nevirtuala(fie B o clasa de baza si D1 si D2 clase derivate din B. Atunci cand D3 deriva din D1 si D2, clasa D3 are elemente in dublu exemplar din B)
B. prin compunere(poti sa ai doua obiecte din doua clase diferite, care mostenesc aceeasi clasa de baza)
C. mostenire si compunere (a aceleasi clase de baza)

14) Descrieti mostenirea virtuala si scopul in care aceasta este folosita :

Mostenirea virtuala este o metoda de implementare a polimorfismului de executie.

-mostenirea virtuala este necesara cand avem 2 clase derivate din clasa de baza si o a treia care mosteneste cele 2 clase(apare problema diamantului – anumite elemente sunt mostenite de doua ori, ceea ce conduce la ambiguitate).

15) Enumerati 3 metode de implementare a polimorfismului de compilare

-supraincarcarea functiilor/operatorilor

-template-uri (sabloane)

-parametrii cu valori implicite

16) Descrieti pe scurt comportamentul operatorului dynamic_cast<>

-se poate aplica pe pointeri sau referinta

-realizeaza convertirea unui obiect de un anumit tip la alt tip.

Sintaxa:

dynamic_cast<tipul la care vrem sa convertim>(ceea ce vrem sa convertim)

17) Descrieti diferenta dintre un pointer si o referinta

POINTER *	**REFERINTA &**
-pointeaza(arata) catre un obiect -obiectul spre care pointeaza poate fi schimbat -pointerul declarat de un anumit tip nu mai poate pointa catre un alt tip.	- este un pointer constant care se deferentiaza automat(un alt nume pentru un obiect) -odata atribuita unui obiect nu mai poate fi schimbata -prin modificare referintei se schimba si obiectul referit.

SUBIECT 2008-VARA

18) Descrieti mecanismul de tratare al exceptiilor

Motivatie: tratarea unitara a exceptiilor, automatizare a exceptiilor

-sunt folosite cuvintele cheie try, throw si catch.

Se implementeaza in felul urmator:

-semnalizarea aparitiei unei exceptii prin intermediul unei valori care semnifica eroarea respectiva

-receptionarea valorii prin intermediul careia se intransmite eroarea

-tratarea(rezolvarea) exceptiei semnalizate si receptionate.

sintax:

try

{ // instructiuni se executa pana la posibila aparitie a erorii

if(test) throw valoare

//instructiuni care se executa daca nu survine eroarea

catch(tip valoare)

{ // instructiuni pentru tratarea/rezolvarea exceptiei }

OBS: Daca aruncam o anumita valoare si nu exista un catch care sa prinda valoarea respectiva , atunci vom avea eroare de executie „Unhandled exception”. Aceasta problema se poate rezolva printr-un catch care prinde orice tip de valoare:

catch(...)(fara tip)

SUBIECTE 2009

19) Descrieti cum se comporta desconstructorii la mostenire:

- ordinea de executie a destructorilor este inversa ordinii de executie a constructorilor(adica de la clasa derivata la clasa de baza). In cazul mostenirii multiple, desconstructorii se apeleaza de la dreapta la stanga in lista de derivare.

20) Descrieti cum este implementat mecanismul de control al tipul in timpul executiei (RTTI)

RTTI= Runtime Type Information

RTTI cuprinde:

A) operatorul typeid : permite aflarea tipului obiectului la executie atunci cand ai doar un pointer sau o referinta catre acel tip.
exemplu:
int myint=50;
cout<<typeid(myint).name();

B) operatia dynamic_cast<>

-se poate aplica pe pointeri sau referinta

-realizeaza convertirea unui obiect de un anumit tip la alt tip.

Sintaxa:

dynamic_cast<tipul la care vrem sa convertim>(ceea ce vrem sa convertim)

21) Descrieti diferenta dintre o clasa si un obiect

CLASA	**OBIECT**
-tip abstract de date, care contine campuri de date(structuri de date) si metode. -nu se aloca memorie atunci cand este creata -este creata folosind cuvantul cheie class	-este o instanta a unei clase. -se aloca memorie atunci cand este creat.

22) Descrieti crearea dinamica de obiecte

-obiectele se pot crea dinamic folosind instructiunea **new** , care aloca memorie in timpul executie in zona de heap.

- constructorul initializeaza dona de memorie alocata.

-eliberarea memoriei se face cu ajutorul instructiunii delete.

- cand un obiect este creat dinamic, metodele si campurile sale se acceseaza cu „->” in loc de „ .”

23) Descrieti functiile sablon si dati exemplu de 3 situatii in care acestea NU genereaza o versiunea a functiei dintr-un sablon disponibil pentru functia respectiva.

o clasa templete este un sablon utilizat pentru generarea unor clase concrete, care difera prin tipul anumitor date membre

-este o metoda de implementare a polimorfismului de compilare

Sintaxa:

template<class Tip>class nume-clasa{ ....}

Odata ce am definit o clasa template, putem crea un exemplar al acesteia folosind:

nume-clasa<tip obiect>ob;

3 situatii in care un apel de functie nu genereaza o versiune a functiei dintr-un sablon:

-cand functia sablon are in lista de parametrii doi parametrii de tipul sablonului, iar noi apelam functia pentru 2 tipuri de date diferite.

-cand functia sablon are un singur parametru de tipul sablonului si mai exista o functie definita cu acelasi nume si cu tip specificat, atunci se va executa functia cu tipul deja specificat.

-cand functia sablon are in lista de parametrii 2 parametrii de tipul sablonului si noi apelam functia pentru un tip de date si un pointer la acel tip.

24) Descrieti diferenta dintre polimorfismul de compilare si cel de executie .

POLIMORFISMUL DE COMPILARE	**POLIMORFISMUL DE EXECUTIE**
-se decide la compilare	-se decide la executie(in functie de anumite informatii disponible la momentul executie)
Curpinde:	**Cuprinde:**
-supraincarcare functiilor/operatorilor	-instantierea dinamica
-template-uri(sabloane)	-mostenirea
parametrii cu valori implicite	-metodele virtuale

25) Descrieti in ce consta mecanismul de incapsulare

Mecanismul de incapsulare este procesul prin care se creeaza un nou tip de date(abstract) definind o clasa ca fiind formata din campuri de date(structuri de date) si metode.

Fiecare camp sau metoda are un atribut de acces:

-public : poate fi accesat de oriunde din program

-protected : poate fi accesat din clasa de baza si din clasele derivate

-private : poate fi accesat doar din clasă.

OBS: Daca nu este pus un atribut de acces, atunci implicit acesta este private:

Exemplu:

```
class A
{
private:
  int x, int y
public: A(){};
}
```

SUBIECTE 2010

26) Descrieti constructorul de copiere

-este un constructor cu un singur parametru de tipul clasei.

-forma implicita(data de compilator) face copierea bit cu bit a informatiei

Utilizare:

-pentru initializarea obiectelor din alte obiecte de acelasi tip

-la transmiterea parametrilor unei functii prin valoare

-la intoarecerea rezultatului unei functii prin valoarea.

27) Cum se pot supraincarca operatorii ca functii independente in C++

Supraincarcarea operatorilor se poate face ca functii membre sau ca functii independente, friend.(pentru a putea avea acces la campurile private ale clasei).

In cazul supraincarcarii ca functie independenta:

-nu avem pointerul *this.

-avem nevoie de toti operanzii ca parametru(pentru ca nu mai exista obiect implicit)

sintaxa:

tip retunrat nume clasa::operator #(lista argumente)

{ //supraincarcare operator }

Exemplu:

supraincarcarea operatorului „<<” pentru afisare sau „>>” pentru citire.

```
friend ostream &operator<<(ostream&os, const clasa& p);
```

OBS: Nu se pot supraincarca ca functie friend operatorii:

„ =” „ ()” „[]”

28) Descrieti in ce consta polimorfismul de executie folosind metode virtuale:

Mostenirea virtuala este o metoda de implementare a polimorfismului de executie.

-mostenirea virtuala este necesara cand avem 2 clase derivate din clasa de baza si o a treia care mosteneste cele 2 clase(apare problema diamantului – anumite elemente sunt mostenite de doua ori, ceea ce conduce la ambiguitate).

29) Descrieti cele doua feluri de folosire a cuvantului „virtual” la mostenire si in ce cazuri se folosesc:

A) **Mostenirea virtuala se realizeaza folosind cuvantul cheie virtual.**

Exexplu:

Class D: virtual public A

Utilitate:

pentru a rezolva ambiguitatea diamantului(doua clase deriva din clasa de baza si avem o a treia clasa care deriva din cele doua clase).

B) **Cand definim functiile utilizand cuvantul cheie virtual, pentru a le putea redefini in clasa derivata**

Exemplu:

```
class A
{
  int x;
public:
virtual int sum(){ //definitie functie }; // functie initiala
}
class B:public A
{
        int y;
 public:
 virtual int sum(){//redefinire functie };
}
```

30) Cum se poate supraincarca operatorul [] si care este utilizarea uzuala a acestuia.

Operatorul [], alaturi de operatorul () si operatorul „=” NU se pot supraincarca folosind functie friend. Acestia trebuie supraincarcati ca functii membre

-este considerat operator binar;

-operatorul [] poate fi folosit si la stanga unei atribuiri(obiectul intors este referinta)

Exemplu:

int v[100]

int & operator[](int i){return vector[i]}

31) Ce este lista de initializare a unui constructor si care este utilitatea ei.

-apare in implementarea constructorului, intre antentul si corpul acestuia.

- lista contien operatorul „ : ” urmat de numele fiecarui membru si valoarea acestuia, in ordinea in care memebrii apar in definita clasei

Utilizare:

-initializarea variabilelor statice

-initializarea referintelor

-initializarea campurilor pentru care nu exista un constructor implicit.

-initializare membrilor clasei de baza

-din motive de eficienta.

Exemplu:

```
 class A
{
int x, y;
public: A(int x, int y)::A(7,9){};
}
```

VARIANTE 2013

32) Sa se scrie diferenta dintre un pointer catre un obiect constat si un pointer constat catre un obiect:

POINTER CATRE OBIECT CONSTANT	**POINTER CONSTANT CATRE OBIECT**
- valoarea obiectului nu poate fi modificata, dar putem modifca obiectul catre care arata pointerul. SINTAXA: const tip de date * p	**-**pointerul nu poate fi modifica, dar valoarea catre care pointeaza da SINTAXA: tip de date const*p

IN 2014 SE REPETA INTREBARILE DE PANA ACUM

MULT SUCCES! 😊 😊 😊