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)