**Declararea obiectelor**

In C++, la fel ca in Java, datele de tip primitiv pot fi doar declarate si abea mai apoi instantiete, adica

**int a;**

**a = 20**

nu ar da nici-o eroare. Totusi, desi in Java merge, in C++ acest lucru nu e valabil si pentru orice tip de obiecte. In C++, un obiect odata ce este declarat, el din start isi aloca memorie si alege zona de memorie pe care sa o aleaga. De aceea, o declaratie de genul:

**TipObiect NumeObiect;**

Nu ar insemna ceea ce inseamna si in Java. In Java, o asa declaratie doar declara un obiect, nu-i aloca memorie, caci asta se face doar cu cuvantul keye new. In C++, variabilele obiecte isi aloca memoria chiar la declarare, deci o astfel de declaratie ar fi valabila doar cu un constructor default. In C++, o variabila obiect, odata declarata, nu mai poate sa schimbe locatia cu care este legata, nici de cum!Desi in Java si Python asta mere merge, dar nu si la datele primitive!Ele aleg un spatiu de memorie si nu il mai pot modifica, ci doar sa copie valorile pe care le primesc in acel spatiu de memorie.

Operatorul **new** se foloseste doar atunci cand dorim sa cream un obiect care sa nu fie distrus decat la finalul programului sau la utilzarea lui **delete**. Acest operator creaza obiectul si returneaza locatia sa, iata de ce o declaratie ca:

Type obj1 = new Type();

ar da eroare! Variabila obj1 vrea un obiect, nu o locatie! In schimb, putem folosi:

Type obj1 = Type();

care va crea un obiect si apoi obj1 il va copia.

Daca pur si simplu cream un obiect ca mai sus, el va fi sters odata ce programul va ajunge la a doua acoala, din functie, de exemplu:

void func(){

Test obj1;

}

Obj1 va fi sters deodata dupa ce se ajunge la }. In schimb, cu new obiectul nu este sters la ajungerea la }

**Functii Virtuale**

Atunci cand o clasa mosteneste membri de la alta clasa, la crearea unui obiect de tip clasa derivata, intai de toate in namespaceul obiectului se va crea un alt namespace in care se vor salva toti membrii ce sunt mosteniti de la clasa parinte si se vor crea obiecte caracteristice de care se vor lega, iar namespaceul de baza al obiectului va avea o referinta la el. Apoi in namespaceul de baza incep sa se adauge membrii caracteristici obiectului. Daca obiectul mosteneste anumiti membri care deja se gasesc in namespaceul sau principal, atunci membrii clasei parinte vor ramane in namespaceul lor, si in namespace principal la obiect se va crea membrul cu referinta la noua sa valore. Cand un membru al obiectului se cheama, el intai este cautat in namespaceul principal, si daca nu e gasit, e cautat in cel mostenit. Functiile virtuale fac ca atunci cand in namespaceul de baza, functia virtuala se leaga de obiectul functie, ea creaza un obiect referinta la obiectul sau functie si il trimite catre functia identica ca prototip din namespace secundar, iar aceeasta se va lega deja de obiectul trimis prin referinta, astfel incat abandoneaza obiectul functie existent. Chiar daca functia virtuala din clasa parinte nu e suprascrisa in clasa copil, functia va pastra in namespace de baza obiectul sau original si la apelarea ei, obiectul o va cauta in namespaceul de baza, nu o va gasi si o va cauta in cel mostenit, o va gasi si o va afisa. Daca un obiect pointer de tip parinte are referinta la copil, el poate accesa doar membrii din namespace secundar al obiectului copil, adica cel ce corespunde parintelui,

**Supra Definirea Operatorilor**

* Operatorul trebuie mereu definit in clasa obiectului din stanga sa

De ex a + b va cere supradefinirea lui + in clasa lui b

cout<<a; ar cere supradefinirea lui << in clasa lui cout, dar nu e tocmai comod, de aceea << se supradefineste intr-o functie globala sau friend

**Obiecte Membrii ai claselor**

* In C++, clasele nu functioneaza la fel ca functiile. Intr-o functie putem usor sa cream un obiect si direct sa il instantiem. De ex:

**Typeobj obj(variabile)**

In clase operatorul () nu poate face referinte la constructorii definiti de user. () se refera in clase doar la metode. Iata de ce,codul de mai jos va da eroare:

class test1

{

public:

test1(int a) {}

};

class test2

{

public:

test1 obj(10);

test2(int a, int b)

{

}

};

Eroarea va aparea din cauza ca clasa poate sa apeleze la definirea lui obj doar constructorul sau default. **Ea va crede ca test1 obj(10) este definirea unei metode, nu a unui membru obiect!Mai exact definirea unui prototip de metoda.**Totusi, daca am scrie :

**test1 obj = test1(10);**

clasa deja va intelege ca noi apelam functia constructor a clasei test1.

Mai putem sa instantiem obiectul in constructor:

class test2

{

public:

test1 obj;

test2(int a, int b) : **obj(a)**

{

}

};

Asa e mai logic, caci constructorul asta si face, ofera anumite valori membrilor, de aceea clasa stie ca daca obj nu a fost inca initializat, va trebui sa astepte ca constructorul sa il intializeze, asa cum test1 nu are constructor default, obj stie ca constructorul il va initializa. Totodata, desi o astfel de declaratie ca test1 obj; ar da eroare intr-o functie, in clasa, intai de toate se adauga membrii in namespace si isi creaza propriul bloc de memorie,dar aceste obiecte ce nu se pot initializa se lasa pe mana constructorului, apoi se executa constructorul si le initializeaza. Adica se va efectua automat instructiunea:

**test1 obj = test1(a)**

Intr-o metoda o asa declarative ca test1 obj; ar da eroare, deoarece obiectul nu poate fi apelay de constructor, si intr-o functie nu are cine sa il initializeze, asa cum constructor nu exista.

* Chiar daca se mosteneste o clasa, apelarea constructorului clasei parinte nu va insemna si initializarea unui membru de tip clasa parinte din clasa copil.

class test1

{

public:

test1(int a) {}

};

class test2: public test1

{

public:

test1 obj;

test2(int a, int b) : obj(a),test1(100)

{

}

};

test1(100) se ocupa de obiectul nou creat, nu de toate obiectele de tip test1 membri.

* In C++ orice variabila declarata,fie ca e instantiate ori nu, intai isi aloca un spatiu de memorie si apoi face o copie daca este egala cu ceva. De ex:

test2 obj1(10,15);

test1 obj2 = obj1

variabila obiect obj2 intai va aloca un spatiu de memorie necesar,dar nu va adauga nimic in el, fara a apela constructorul, apoi va copia tot ce are obj1. Ea nici nu are nevoie sa apeleze constructorul. Mai mult ca atat, un = intre 2 obiecte de acelasi fel cheama constructorul copy, deci tip obj1 = obj2(sunt de acelasi fel) e similar cu tip obj1(obj2)

* Dar, dat fiind ca test1 are doar un constructor cu un parametru, o declaratie ca

test1 obj;

intr-o functie ar da eroare, deoarece nu exista un constructor ce sa fie apelat si nu are cine sa il apeleze pe obj, nu exista doar constructor ce ar putea sa o faca.

**Static**

Static inseamna ca ca o var sau un obiect se initializeaza o singura data si este sters doar la incheierea programului.

* O variabila statica dintr-o functie este o variabila pentru care se aloca o singura data memorie, chiar daca functia este apelata de mai multe ori.

void func()

{

static int a = 0

++a

}

Deci lui a I se va aloca memorie o singura data, adica la prima declarative a functiei, dupa la fiecare apel valoarea ei doar se va incrementa, nu se va seta iar la 0, caci ea nu poate fi din nou create.

* Variabilele statice dintr-o clasa sunt puse intr-un namespace special impartasit de toate obiectele de tipul acelei clase. Ele nu pot fi modificate in constructori!
* Metodele statice din clasa pot apela doar membrii statici, ca in Java, sau ca in Python cum o fac metodele de clasa. Ele nu memorizeaza starea variabilelor din ele.
* O functie statica in C doar face ca ea sa nu fie accesibila din alt fisier.
* This nu poate fi utilizat in metode statice

**Exceptii**

**Checked(nu exista in C++) –** se verifica chiar in timpul compilarii programului. Spre exemplu, FileReader sau FileInputStream cer ca exceptiile ce ar putea sa le genereze sa fie mereu administrate in cod.

**Unchecked –** se verifica chiar cand programul ruleaza.

* Toate exceptiile in C++ sunt unchecked
* catch(…) – pentru orice exceptie ce nu are un bloc catch
* finally nu exista in C++
* Putem arunca si prinde o eroare de orice tip,integer,string etc.
* Dup ace blocul try se executa sau daca trebuie sa arunce ceva, toati membrii din blocul try sunt distrusi
* Daca avem, de exemplu, 2 obiecte, si al 2 obiect mosteneste de la primul obiect, atunci blocurile throw, daca vrem sa se ocupe de ambele, trebuie in puse in ordinea de la obiectul copil la cel parinte. Adica, intai putem un catch pentru un obiect de tip copil, apoi de tip parinte. De altfel, daca intai merge un bloc catch pentru parinte si apoi pentru copil, blocul pentru copil nu va fi niciodata executat, indifferent ce obiect e aruncat, din cauza ca obiectul copil e si obiect de tip parinte, deci ca si cum copilul deja e prins de parinte. In Java compilatorul nici nu ne-ar lasa sa face asa ceva.Mereu va cere sa o facem in ordine.
* Ca si in Java, daca o functie intalneste o exceptie, ea va cauta blocul catch corespunzator chiar si de unde a fost apelata. Daca in try apelam o functie, ea deja poate folosi throw si fara un try, asa cum ea deja se afla intr-un try. Obiectele din functie sunt distruse in ordine inversa deci. La fel e si in Java.
* Pana la **C++17**, se putea seta ce tip de exceptii ar putea sa arunce o functie.De ex:

Void func(int a) throw(int,int \*)

{

}

Totusi, aceasta declaratie era problematica. Daca se arunca totusi un tip de exceptie nespecificat in throw, functia s-ar fi putut comporta neasteptat. Se putea pune si throw() pentru a spune ca functia nu arunca nimic, dar daca arunca totusi, iar se putea purta programul neasteptat.

Totusi, in C++17, throw() = noexcept=noexcept(true). noexcept sugereaza ca functia nu arunca nimic. Utilizarea lui throw(int,int\*), de ex,mai e permisa pana la C++ 14, dar ea oricum va fi echivalenta cu noexcept(false)

* noexcept(false) – functia poate arunca o exceptie de orice tip

**Lucrul cu Fisierele**

ifstream – doar citire

ofstream – doar scriere

fstream – scriere si citire

* **stream** – secventa de biti primita de program si trimisa de program
* Pentru a deschide un fisier, putem crea un obiect de aceste tipuri si sa trimitem constructorului adresa fisierului + optional modul de lucru prin **ios::**. Sau pur si simplu sa folosim metoda **open()** daca nu folosim constructorul
* Obiectele ifstream au automat setat **ios::in** iar cele ofstream au automat setat **ios::out** iar cele fstream au **ios::in | ios::out**
* Ifstream poate citi fisierele prin operatorul >>(va citi cate un singur cuvant) sau functiie read(),get(),getline(),seekg(),tellg()
* Ofstream poate scrie prin operatorul << sau prin metodele put(),write(),seekp(),tellp()
* **seekg**(pozitie in string din file,ios::poztiei buffer) – seteaza bufferul de citire la pozitia initializata.

ios::begin – de la inceputul fisierului

ios::end – de la finalul fisierului

ios::cur – se ia de la pozitia curenta din buffer

* **tellg**() – returneaza pozitia curenta din buffer de citire
* **seekp**(pozitie) – seteaza bufferul de scriere la pozitia data
* **tellp**() – returneaza pozitia curenta din bufferul de scriere

**Destructor Virtual**

**Friend class si functie**

**Typedefine**

**Const**

* Daca e pus la inceput de declarare de functie, inseamna ca va returna un tip de date const. De ex:

**const int func(){**

**…….**

**}**

Functia data va returna un tip de date **const int,** dar nu inseamna ca ea e constanta!

* Daca e pusa dupa functie, inseamna ca functia data e constanta si nu se pot modifica membrii clasei in ea!

class clasa1{

Int a = 10;

void func(){ this.a = 100} #**ERROR**!

}

* Obiectele constante pot accesa doar functiie constante din clasa!
* O variabila constanta trebuie initializata exact in momentul crearii, daca e membra a unei functii.
* Pointerii tot pot folosi const:

int const \*ptr sau const int \*ptr - pointer catre o var int constanta

int \* const ptr – pointer constant catre o var int

const int \* const ptr – pointer constant catre o var int constanta

* Daca un membru al unei clasei e constant, exact ca si in Java, el poate fi initializat fie la crearea ei, fie de un constructor.Constructorul il initializeaza asa:

class clasa1{

public:

const int a;

clasa1() : a(valoare) { }

};

* Totusi, chiar daca un membru constant are deja o valoare initializata, construtorul poate sa o modifice.

class clasa1{

public:

const int a = 100;

clasa1() : a(valoare\_noua) { }

};

**Namespace**

* **Namespace** – loc special unde sunt pastrati identificatorii(varibile,clase,metode etc).
* Structura

namespace nume{

variabile….

functii…

clase….

}

* Pentru a-l utiliza folosim **nume::identificator**;
* **using namespace** plaseaza toate elementele din acel namespace in namespace de baza al programului.
* Identificatorii dintr-un namespace nu pot fi utilizati fara name:: pana nu se pune using namespace.
* **Compilatorul nu va cauta niciodata automat in namespaceuri existente in cel de baza!Cam asa cum face in clase.**
* using namespace poate fi utilizat in alte namespace sau in alte functii/metode, insa nu poate fi utilizat in clase.

namespace space {

int a = 100;

}

int main() {

using namespace space;

cout << a;



return 0;

}

* Un namespace poate fi declarat doar in namespace principal al programului sau in alt namespace din acesta.

int main() {

namespace space {

int a = 100;



}

using namespace space;

cout << a;

return 0;

}

* Pentru a accesa un element dintr-un nested namespace, folosim

namespace1::namespace2::cod;

* Daca folosi using namespace si in namespace principal se copie o variabila care deja exista in el, se va crea o ambiguitate si va aparea eroare. Acelasi lucru e valabil si daca doua namespace au aceeasi variabila si se adauga ambele cu using nemaspace;

namespace space {

int a = 100;

}



using namespace space;

int a = 1;



int main() {

cout << a;



return 0;

}

* **Un namespace poate fi redefinit, adica putem crea mai multe namespace cu acelasi nume, caci oricum compilatorul il va lua ca unul singur, dar nu putem sa redeclaram variabile deja existente!**



namespace space {

int a = 100;

}

namespace space {



int a = 100;



int b = 1;

}



using namespace space;

int main() {

cout << a << endl;

cout << b;

return 0;

}

* Putem sa utilizam doar **using** fara **namespace,** si sa scriem namespace::cod, adica asa:



using nume::cod;

asa se evita copierea tuturor variabilelor din namespace in cel de baza, si se copied oar cel specificat. Restul trebuie sa fie accesate cu namespace::cod;

namespace space {

int a = 100;

int b = 200;

}

using space::a;



int main() {

cout << a << endl;

cout << space::b;

return 0;

}

* Putem crea o clasa intr-un namespace si apoi sa o instantiem.



namespace space {

class clasa {

public: int a = 10;

};

}

int main() {

space::clasa obj;

cout << obj.a;

return 0;

}

* Putem redefini clase sau metode din namespaceuri

namespace space {

int func();

class clasa {

public:

int func();

};

}

int space::func(){ }

int space::clasa::func() {}

int main() {

return 0;

}

**Arrays**

* Nu putem crea arrays cu new!

~~int a[] = new int[5];~~

* O matrice mereu va cere ca sa se specifice numarul de elemente din fiecare tabel, nefiind posibil ca fiacre tablou din matrice sa aiba propriul numar de elemente.

~~int a[][];~~

* Doar arrays globale, la declarare, adauga automat 0 la toate elementele lipsa,fie ca e declarant vreun membru din array sau nu, si cele ca membrii ai clasei, ca in Java. In metode sau functii, ca sa se adauge 0 la elementele lipsa, e necesar sa scriem cel putin un membru din arrays la declararea lui.

int a[5];



int main() { return 0}

class a{

public:

int a[5];



};

class a{

public:

void func(){

int a[5];



int a[0] = 1;



int b[5] = {1};



}