Basic concepts of OOP in C++:

* Classes & Objects
* Abstraction
* Encapsulation
* Inheritance
* Polymorphism

An [Object](https://www.simplilearn.com/tutorials/cpp-tutorial/class-and-object-in-cpp) can be defined as an entity that has a state and behavior, or in other words, anything that exists physically in the world is called an object.

Un **program orientat obiect** este o colectie de obiecte care interactioneaza unul cu celalalt aplicand o metoda

[Class](https://www.simplilearn.com/tutorials/cpp-tutorial/classes-in-cpp) can be defined as a blueprint of the object. It is basically a collection of objects which act as building blocks. Are by default: un constructor de initializare, unul de copiere, un destructor, un operator de atribuire.

**class** employee {

**char** name[80]; // Obs: variabilele nestatice nu pot fi initializate aici la declarare

**double** wage;

**public**:

**void** putname(**char** \*n);

**void** getname(**char** \*n);

**void** putwage(**double** w);

**double** getwage();

};

[Abstraction](https://www.simplilearn.com/tutorials/cpp-tutorial/abstract-class-in-cpp) helps in the data hiding process. It helps in displaying the essential features without showing the details or the functionality to the user.

Encapsulation is the wrapping up of data and functions together in a single unit and makes the data non-accessible to the outside world -> scope: private, public, protected, virtual

**Obs:** All members of a class are by default **private** in C++ ! .

Member variables have to be private! .

Membrii privati din clasa parinte nu sunt vizibili in clasele derivate .

Membrii protected sunt vazuti doar de clase derivate sau friend .

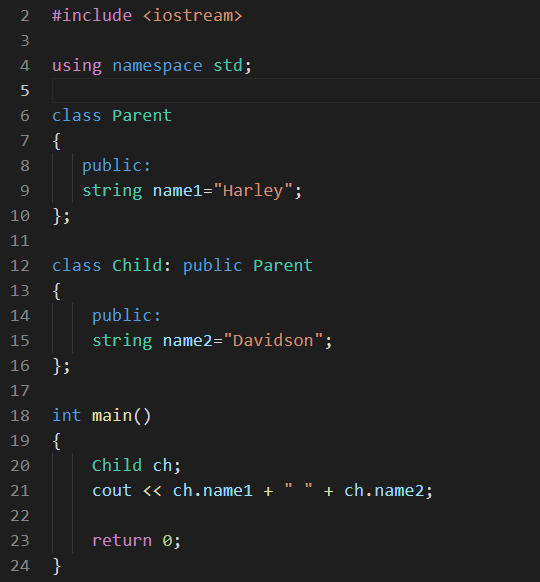
Obs: **struct** si union au default membri ca public, iar class ca private.

Functiile **friend** se folosesc pentru accesarea campurilor protected, private din alta clasa, folositoare la overload-area operatorilor, pentru unele functii de I/O, si portiuni interconectate (exemplu urmeaza), in rest nu se prea folosesc (pt acces la campuri se fol getterii si setterii).

Clasa prieten: toate functiile membre ale clasei prieten au acces la membrii privati ai clasei.

Metodele virtuale sunt evaluate la EXECUTIE (din clasa din care face parte obiectul -care are atasata informatie suplimentara).

Metodele nevirtuale se evalueza la COMPILARE (din clasa din care face parte variabila prin care se acceseaza -pointer, referinta, obiect )



 Inheritance:

Initially, you created an instance of the child class, through which you are calling name1 and name2 variables.

Since child class is inheriting parent class, when you call using the object of the child class, you get the result name1 as Harley and name2 as Davidson.

[Polymorphism](https://www.simplilearn.com/tutorials/cpp-tutorial/polymorphism-in-cpp) means the ability to take more than one form. It is a feature that provides a function or an operator with more than one definition. It can be implemented using function overloading, operator overload, [function overriding](https://www.simplilearn.com/tutorials/cpp-tutorial/function-overriding-in-cpp), virtual function.

Ex: Classes Animal, Cheetah and Dolphin that have a[function](https://www.simplilearn.com/tutorials/cpp-tutorial/cpp-functions) of the same name (public: void speed() { } ), but the definition of this speed function is different in all three classes.. When you call the function using the object of the child class, it overrides the parent class function and prints the function of the child class.

**Overloading** = Supraincarcare = Compile time polymorphism = Static binding = Early binding

* Utilizarea mai multor funcţii care au acelaşi nume,
* Identificarea se face prin numărul de parametri şi tipul lor. **Tipul de întoarcere NU e suficient pentru a face diferența.**

Exemplu:

**void afis** (**int** a) { cout<<”int”<<a; }

**void afis** (**int** a, int b) { cout<<”char”<<a; }

-Alegerea la apel a unei functi supraincarcate (cu acelasi nume) se face in 3 pasi

pas1. Potrivire exacta

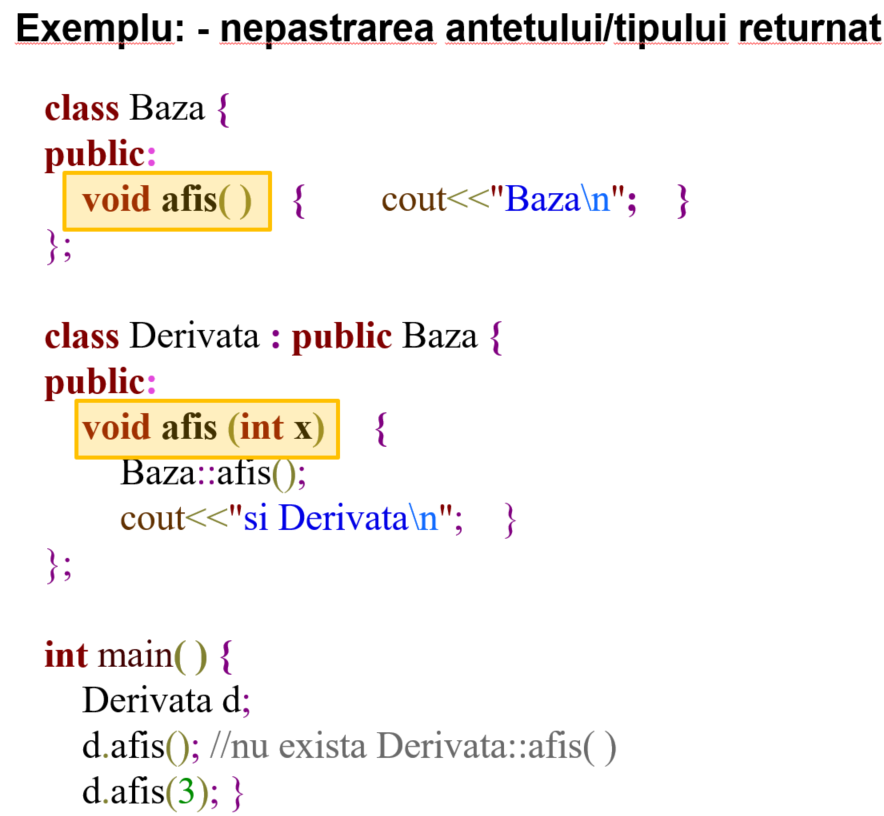
pas2. Conversie fara pierderi

pas3. Conversie cu pierderi

**Overriding** = Redefinire = Run time polymorphism = Dynamic binding = Late binding

* + **cu acelasi antet ca în clasa de baz**ă (“redefining” - pt fct oarecare / “overloading” - pt fct virtuale);
  + **cu schimbarea listei de argumente sau a tipului returnat**.

***Obs: la redefinirea unei funcţii din clasa de baza, toate celelalte versiuni sunt automat ascunse!***



1. ***Run - Time Type Identification (RTTI)*** - permite identificarea tipului unui obiect in timpul executiei programului;
2. set aditional de 4 operatori de cast (***dynamic\_cast ->*** legat direct de mecanismul RTTI***, const\_cast, reinterpret\_cast,***  si ***static\_cast***) - pentru o modalitate mai sigura de cast:

C++ implementeaza polimorfismul prin mostenire, functii virtuale si pointeri catre clasa de baza care pot fi utilizati pentru a arata catre obiecte din clase derivate, deci nu se poate sti a-priori tipul obiectului catre care se pointeaza. ***Determinarea se face la executie, folosind RTTI*** (nu functioneaza cu pointeri void, pt ca nu au informatie de tip)***:***

* ***typeid*** - pentru a obtine tipul obiectului predefinit sau definit de utilizator;
* ***#include <typeinfo>***;
* uzual ***typeid(object)***; - returneaza o referinta catre un obiect de tip **type\_info** care descrie tipul obiectului;

Templates (Sabloane) = when multiple functions do identical operations

* Functii generice (cu unul sau mai multe tipuri generice)
* Clase generice
* Sabloane de proiectare (Design Patterns): Singleton, Abstract Object Factory, Observer => **S.O.L.I.D**

Advantages of OOPs

There are various advantages of object-oriented programming.

* OOPs provide reusability to the code and extend the use of existing classes.
* In OOPs, it is easy to maintain code as there are classes and objects, which helps in making it easy to maintain rather than restructuring.
* It also helps in data hiding, keeping the data and information safe from leaking or getting exposed.
* Object-oriented programming is easy to implement.

 Completari aduse limbajului C++ fata de C

* supraincarcarea functiilor (inclusiv pentru operatii de I/O (<<,>>))
* funcţii cu valori implicite (trb amplasate la sfarsitul listei de argumente)
* alocarea dinamica cu new si delete
* tipul referinta (**trebuie iniţializată in momentul declararii,** dacă nu este membra a unei clase, un parametru de functie sau o valoare returnata)
* functii in structuri
* comentarii pe o singura linie
* citirea string-urilor pană la primul caracter alb
* posibilitate de afișare folosind toate caracterele speciale \n, \t, etc

Tipul Referință

int main() {

int a = 20;

int& ref = a;

cout<<a<<" "<<ref<<endl; // 20 20

ref++;

cout<<a<<" "<<ref<<endl; // 21 21

}

**Obs:**

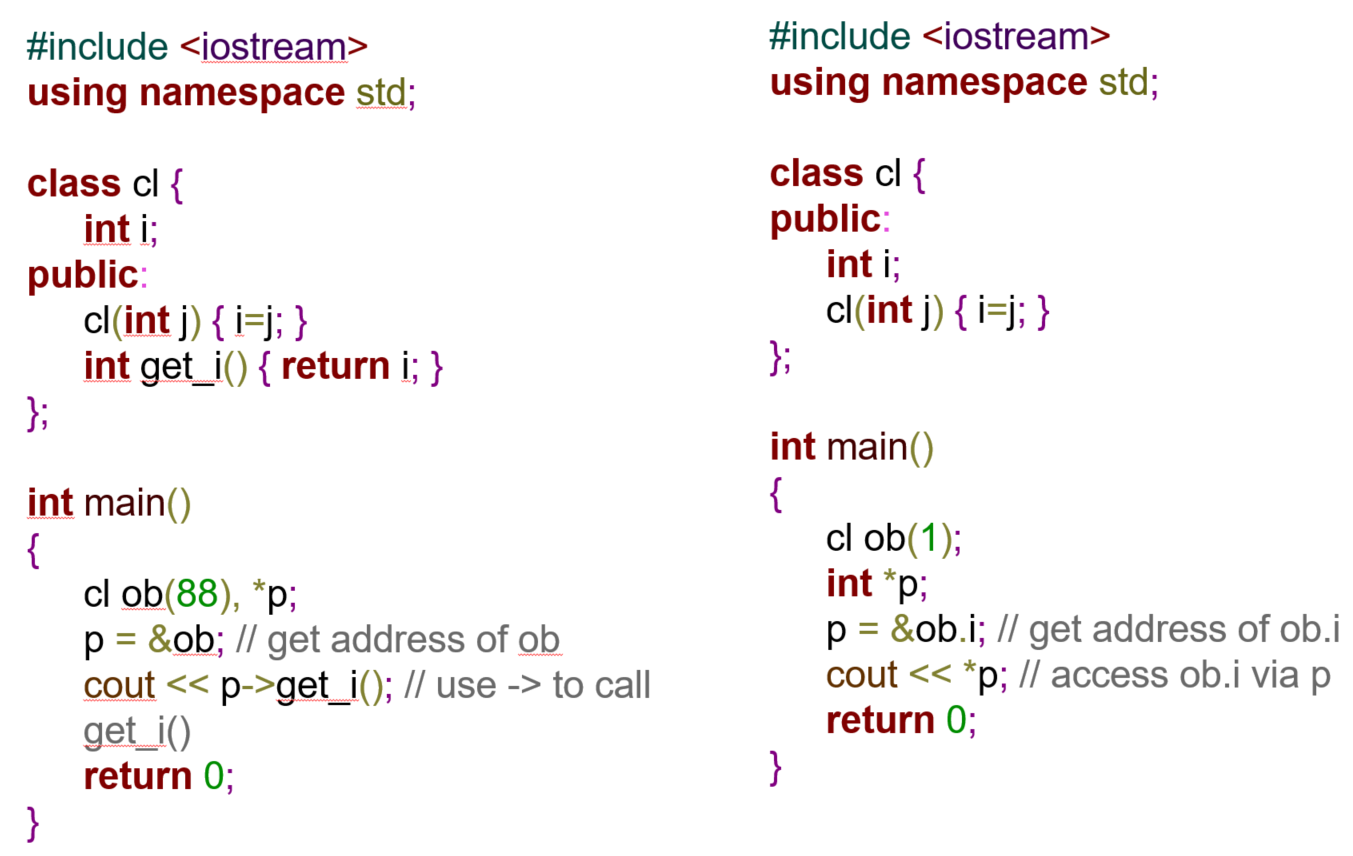
* Pentru a putea fi folosită, o referinţă trebuie iniţializată in momentul declararii, devenind un alias (un alt nume) al obiectului cu care a fost iniţializată.
* spre deosebire de pointeri care la incrementare trec la un alt obiect de acelasi tip, incrementarea referintei implica, de fapt, incrementarea valorii referite.
* in afara initializarii, nu puteti modifica obiectul spre care indica referinta.
* referintele nule sunt interzise intr-un program C++ valid.
* nu se poate obtine adresa unei referinte.
* nu se pot crea tablouri de referinte.
* nu se poate face referinta catre un camp de biti
* dacă transmitem obiecte prin apel prin referință la funcții nu se mai creează noi obiecte temporare, se lucrează direct pe obiectul transmis ca parametru

**Transmiterea parametrilor:**

* parametrii formali - sunt creati la intrarea intr-o functie si distrusi la retur;
* apel prin valoare - copiaza valoarea unui argument intr-un parametru formal ⇒ modificarile parametrului nu au efect asupra argumentului;
* apel prin referinta - in parametru este copiata adresa unui argument ⇒ modificarile parametrului au efect asupra argumentului.

Tipul Pointer

* obiectele sunt în memorie &obiect;
* accesarea membrilor unei clase: -> în loc de .
* tipurile pointerilor trebuie să fie la fel



Pointerul this

* orice funcție membru are pointerul **this** (definit ca argument implicit) care arată către obiectul asociat cu funcția respectivă
* funcțiile prieten si cele statice **nu** au pointerul this

B \*p, o(1);

D oo(2);

p=&o;

p=&oo;

* pt clasa de bază B şi clasa derivată D => un pointer către B poate fi folosit şi cu D;

Operatori:

Operatorii sunt funcții cu un nume special: cuvântul rezervat operator, urmat de simbolul operatorului pe care îl definim. La fel ca orice altă funcție, și operatorii au tip al rezultatului și listă de parametri. Operatorii care pot fi supraîncărcați sunt: +, -, =, \*, /, %, ^, &, |, ~, <, >, <=, >=, ==, !=, !, &&, ||, ++, --, <<, >>, +=, =, /=, %=, ^=, &=, |=, \*=, <<=, >>=, [], (), ->, ->\*, new, new [], delete, delete [] si ,

Următorii operatori **nu** pot fi supraîncărcați:

**. (acces de membru)**

**:: (rezolutie de scop)**

**.\*(acces membru prin pointer)**

**? (ternar)**

Doua moduri de a defini operatori pentru o clasă: prin intermediul metodelor sau al [funcțiilor prietene](https://www.pbinfo.ro/articole/3900/programare-orientata-obiect-introducere#intlink-4).

Putem folosi metodele când operatorul este unar, sau când este binar și primul operand este obiect al clasei pentru care definim operatorul. Dacă operatorul este binar și primul operator nu este obiect al clasei, vom defini operatorul prin intermediul funcțiilor prietene. Obs: nu se pot supraincarca = () [] sau -> cu functii prieten, iar pentru ++ sau -- trebuie sa folosim referinte.

***Operatorii de cast***

1) operatorul traditional mostenit din C;

2) dynamic\_cast;

3) static\_cast;

4) const\_cast;

5) reinterpret\_cast.

***Dynamic\_cast***

* daca vrem sa schimbam tipul unui obiect la executie: **schimba tipul unui pointer/referinte** intr-un alt tip **pointer/referinta**.
* se verifica daca un downcast este posibil (si deci valid);
* daca e valid, atunci se poate schimba tipul, altfel eroare.

Sintaxa: **dynamic\_cast <target-type> (expr)**

* target-type trebuie sa fie un pointer sau o referinta;

Exemplu:

class B{virtual …};

class D:B {... };

* un pointer D\* poate fi transformat oricand intr-un pointer B\* (pentru ca un pointer catre baza poate oricand retine adresa unei derivate);
* invers este necesar operatorul dynamic\_cast.

Base \*bp, b\_ob;

Derived \*dp, d\_ob;

bp = &d\_ob; // base pointer points to Derived object

dp = dynamic\_cast<Derived \*> (bp); // cast to derived pointer OK

bp = &b\_ob; // base pointer points to Base object

dp = dynamic\_cast<Derived \*> (bp); // error

***Static\_cast***

int main() {

int i;

for(i=0; i<10; i++)

cout << **static\_cast<double> (i)** / 3 << " ";

return 0;

}

* este un substitut pentru operatorul de cast clasic;
* lucreaza pe tipuri nepolimorfice;
* poate fi folosit pentru orice conversie standard;
* ne se fac verificari la executie (run-time);

**Sintaxa: static\_cast <type> (expr)**

***Const\_cast***

* folosit pentru a rescrie, explicit, proprietatea de const sau volatile intr-un cast (elimina proprietatea de a fi constant);
* tipul destinatie trebuie sa fie acelasi cu tipul sursa, cu exceptia atributelor const / volatile.

Sintaxa: const\_cast <type> (expr)

void sqrval(const int \*val) {

int \*p;

**// cast away const-ness.**

**p = const\_cast<int \*> (val);**

\*p = \*val \*\*val; // now, modify object through v

}

int main() {

int x = 10;

cout << "x before call: " << x << endl;

sqrval(&x);

cout << "x after call: " << x << endl;

return 0;

}

void sqrval(const int &val) {

**// cast away const on val**

**const\_cast<int &> (val) = val \* val;**

}

int main() {

int x = 10;

cout << "x before call: " << x << endl;

sqrval(x);

cout << "x after call: " << x << endl;

return 0;

}

***Reinterpret\_cast*** - converteste un tip intr-un alt tip fundamental diferit;

Sintaxa: reinterpret\_cast <type> (expr)

int main()

{

int\* p = new int(65);

char\* ch = **reinterpret\_cast<char\*>**(p);

cout << \*p << endl;

cout << \*ch << endl;

cout << p << endl;

cout << ch << endl;

return 0;

}

- La aruncarea exceptiilor NU se face CONVERSIE catre tipul argumentelor din instructiunile catch.

Exceptie: obiecte din Derivata sunt prinse de catch cu argument Baza

Agregarea/Compunerea (‘has a’) vs. Mostenirea (‘is a’)

class **Profesor** { .. };

class Curs {

**Profesor p;**

};

class Building { .. };

**class** School : **public** Building { ..

Mostenirea

class A : **public / protected / private** B { .. };

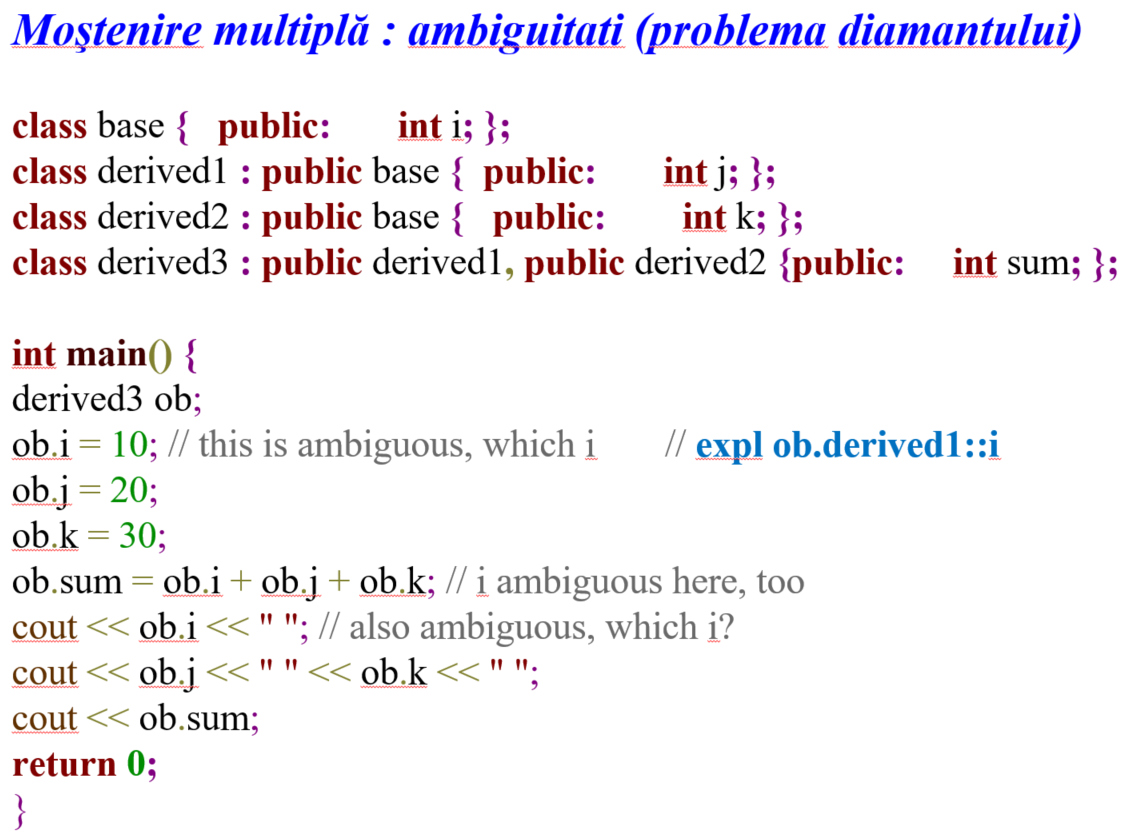
Dacă lipsește modificatorul de acces, atunci e considerat implicit private.

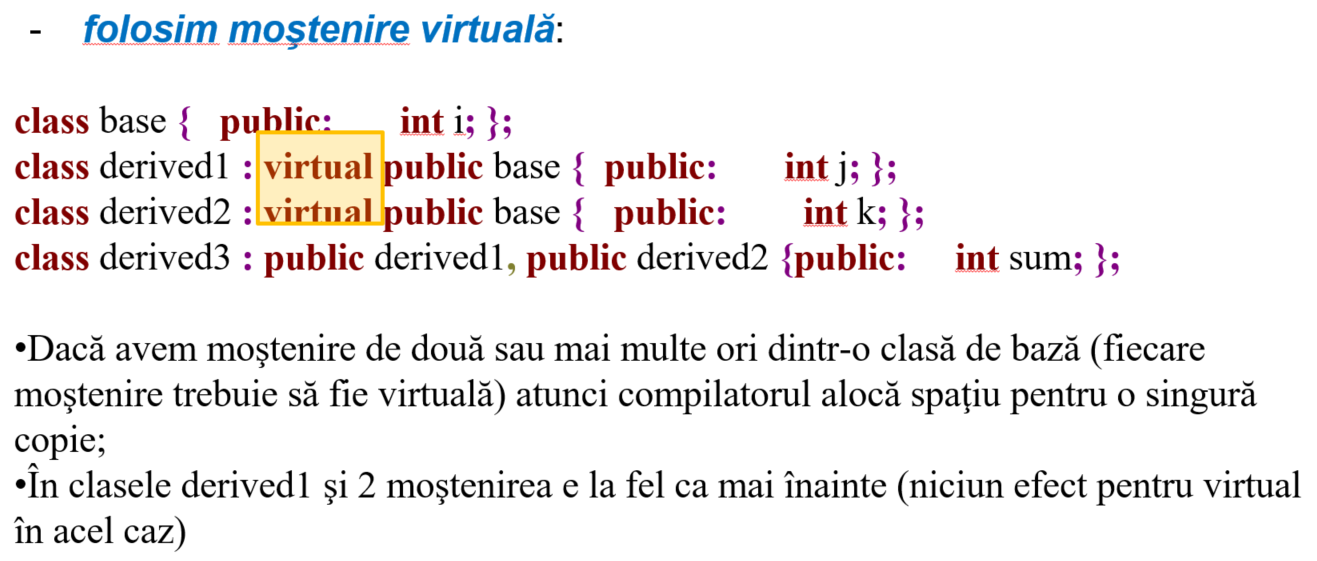
* **public** => membrii din clasa de bază işi păstrează tipul de acces şi în derivată.
* **private** => toți membrii din clasa de bază vor avea tipul de acces “private” in derivată, indiferent de tipul avut în bază !!
* **protected** => membrii “publici” din clasa de bază devin “protected” în clasa derivată, restul nu se modifică (cei protected raman protected)

Obs: este mai bine a se utiliza compunerea în locul moştenirii private;

Compatibilitatea între o clasă derivată şi clasa de bază=> **conversii implicite de tip**:

* + dintr-un obiect derivat într-un obiect de bază;
  + dintr-un pointer sau referinţă la un obiect din clasa derivată într-un pointer sau referinţă la un obiect al clasei de bază

****

****

**Functii virtuale**

* fol pt polimorfism la executie
* definite in baza si redefinite in derivata

Obs: Pointer de tip bază care arată către obiect de tip derivat şi cheamă o funcţie virtuala în bază şi redefinite în clasa derivată executa ***funcţia din clasa derivată***.

**Constructorii clasei derivate**

-Constructorul se apeleaza cand se ALOCA (explicit (new) sau implicit (la declarare)) zona pentru un obiect.

(La alocarea de zona pentru un vector de obiecte se apeleaza constructorul pentru fiecare element al vectorului )

(Nu se apeleaza constructorul la declarare de pointer sau declarare de referinta)

-Constructorul de copiere se apeleaza cand este alocata zona pt un obiect si este si initializat cu datele unui alt obiect deja creat

-Constructorul de copiere se mai apeleaza

- la functiile care transmit parametrii prin valoare (se creaza o copie a parametrului actual)

- la functiile care intorc rezultatul prin valoare (se creaza o copie a rezultatului )

- Constructorul de copiere dat de compilator copiaza in obiectul implicit datele obiectului parametru -bit cu bit

-Operatorul= se apeleaza la atribuire intre 2 obiecte deja create

-Operatorul = dat de compilator copiaza in obiectul implicit datele obiectului parametru -bit cu bit

-Destructorul se apeleaza cand zona alocata obiectului este eliberata (explicit (delete) sau de catre sistem (se iese din blocul in care a fost definit))

-Daca se declara un constructor de initializare -compilatorul nu mai asigura un constructorul de initializare

-Daca se declara un constructor de copiere -compilatorul nu mai asigura nici constructor de initializare si nici constructor de copiere

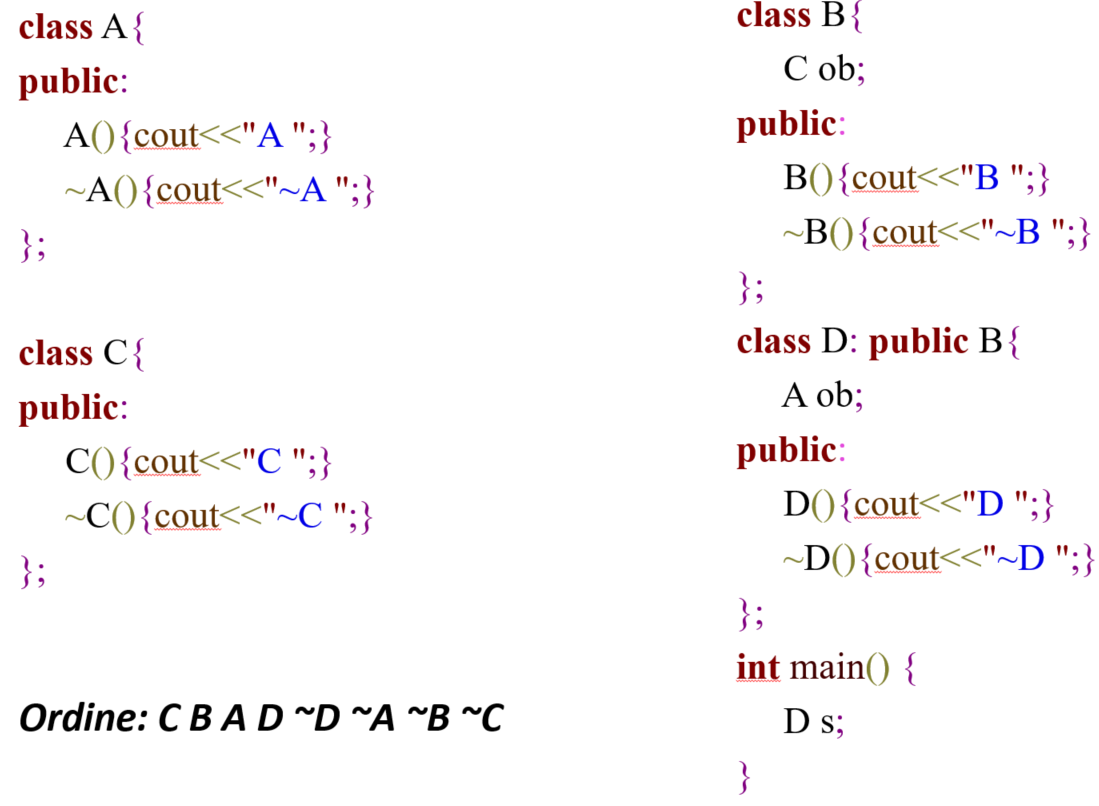
-Pentru crearea unui obiect din clasa derivata -se apeeaza intai constructorul clasei de baza , apoi constructorul clasei derivate (destructorii -invers)

-Pentru crearea unui obiect dintr-o clasa compusa -se apeleaza intai constructorul pentru datele membre si apoi contructorul clasei compuse (destructorii invers)

-Constructorul de copiere dat de compilator in clasa derivata apeleaza implicit constructorul de copiere al bazei

-Constructorul de copiere definit de programator apeleaza implicit constrctorul de INITIALIZARE al bazei

**Ordinea apelării constructorilor şi destructorilor**

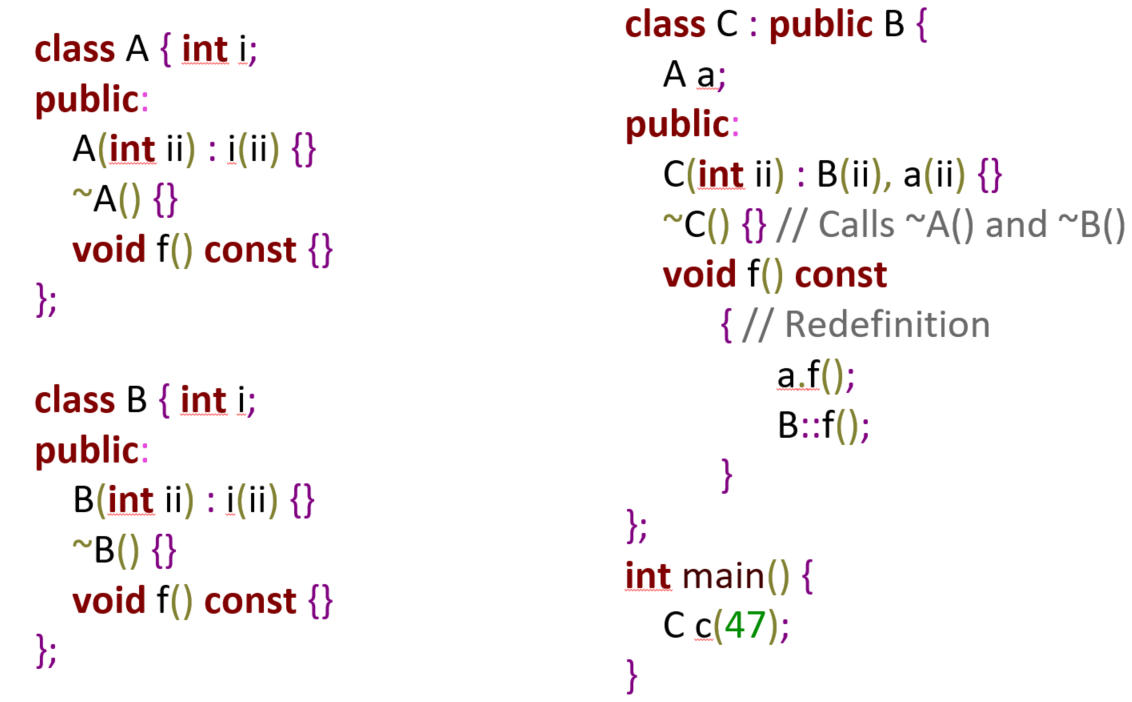


1) In acelasi domeniu de vizibilitate, constructorii se apeleaza in ordinea declararii obiectelor, iar destructorii in sens invers.

2) Variabilele globale se declara inaintea celor locale, deci constructorii lor se declara primii.

3) Daca o functie are ca parametru un obiect, care nu e transmis cu referinta, atunci se activeaza constructorul de copiere si, implicit, la iesirea din functie, obiectul copie se distruge, deci se apeleaza destructor.

***“Pseudo - constructori”:***



class Forma {

protected:

int h;

public:

Forma(int a = 0) { h = a; }

};

class Cerc: public Forma {

protected:

float raza;

public:

Cerc(int h=0, float r=0**) : Forma(h)** { raza = r; }

};

Array-uri de obiecte alocate dinamic

* + nu se pot inițializa
  + trebuie să existe un constructor fără parametri
  + delete poate fi apelat pentru fiecare element din array

**Static**

* date membre **statice** = unice pentru toate obiectele clasei, exista o singura copie pentru toate obiectele
* Sunt create INAINTEA oricarui obiect si initilizate implicit cu 0 (sau explicit).
* create, initializate si accesate – independent de obiectele clasei.
* trebuiesc redeclarate dupa definirea clasei (in afara clasei) pentru a li se aloca zona de memorie.
* functiile statice:

- efectueaza operatii asupra intregii clase => nu au cuv cheie “this” pt ca nu sunt asociate unui obiect

- **se pot referi / pot accesa doar la membrii statici.**

class static\_type {

static int i;

public:

static void init(int x) { i = x;}

void show() {cout << i;}

};

**int static\_type::i = 1;**

Funcțiile membre statice se comportă exact ca și funcțiile nemembre: se moștenesc în clasa derivată.

-Redefinirea unei funcții membre statice duce la ascunderea celorlalte supraîncărcări.

-Schimbarea signaturii unei funcții din clasa de bază duce la ascunderea celorlalte versiuni ale funcției.

**Dar: O funcție membră statică nu poate fi virtuală.**

**Const**

* trb initializat la declarare sau in lista de initializare a constructorului, singura excepție fiind:

**extern const int bufsize;**

=> const implică “***internal linkage***” adică variabila e vizibilă numai în fișierul respectiv (la linkare)

* compilatorul nu alocă spațiu pentru constante, dar dacă e declarat ca extern alocă spațiu (să poată fi accesat şi din alte părți ale programului)
* static const trebuie inițializat la declarare (nu în constructor)
* pentru a se asigura că starea obiectului nu se schimbă funcțiile de instanță apelabile trb def cu const
* metodele const au declaratia const DUPA lista de parametrii.
* functiile membru const nu pot apela funcții non-const si in ele nu se pot modifica obiecte/date

// Constants and aggregates

**const int i[] = { 1, 2, 3, 4 };**

// ! float f[i[3]]; // Illegal

**struct S { int i, j; };**

**const S s[] = { { 1, 2 }, { 3, 4 } };**

//! double d[s[1].j]; // Illegal

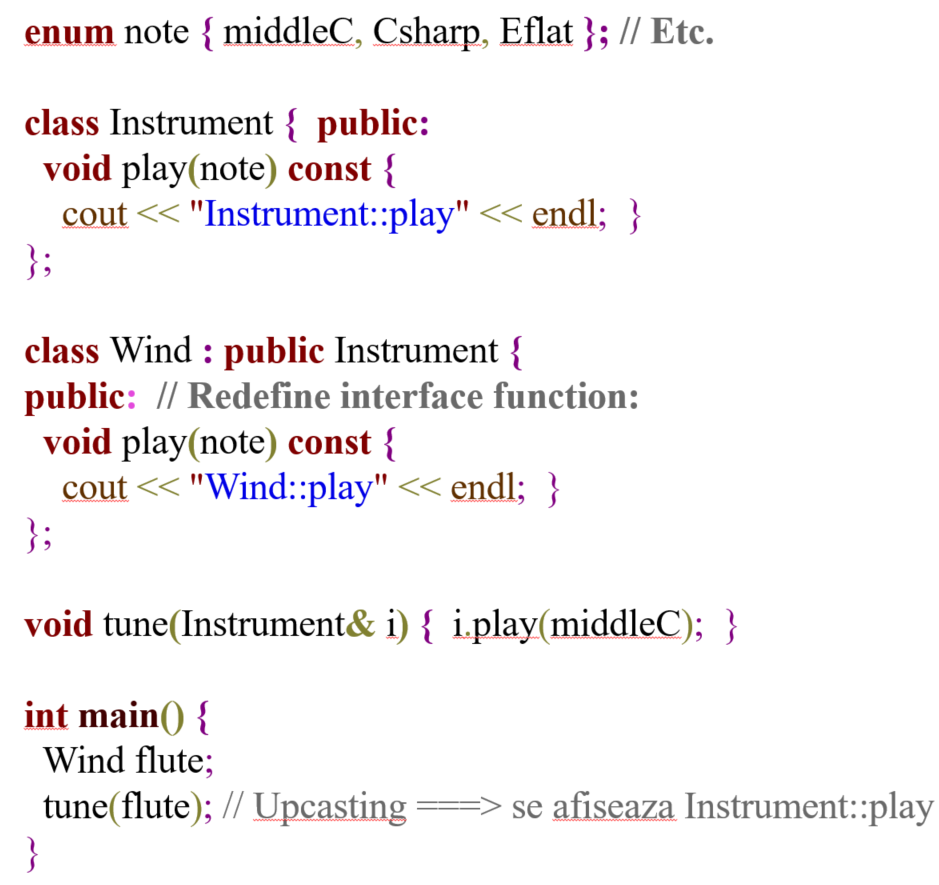
Pointeri const:

* const se alatură elementului cel mai apropiat: const int\* u;
* u este pointer către un int care este const: int const\* v; la fel ca mai sus
* se poate face atribuire de adresă pentru obiect non-const către un pointer const
* nu se poate face atribuire pe adresă de obiect const către pointer non-const

**Upcasting**

- Tipul derivat poate lua locul tipului de bază (f important pentru procesarea mai multor tipuri prin acelaşi cod).

**Funcţii virtuale: ne lasă să chemăm funcţiile pentru tipul derivat.**



***Clase abstracte și funcții virtuale pure***

Clasă abstractă = are cel puțin o funcție virtuală PURĂ

Necesitate: clase care dau doar interfață (nu vrem obiecte din clasă abstractă ci upcasting la ea).

Eroare la instantierea unei clase abstracte (nu se pot defini obiecte de tipul respectiv).

Permisă utilizarea de pointeri și referințe către clasă abstractă (pentru upcasting).

Nu pot fi trimise către funcții (prin valoare).

Obs: La moștenire, dacă în clasa derivată nu se definește funcția pură, clasa derivată este și ea clasă abstractă ---> nu trebuie definită funcție care nu se execută niciodată

UTILIZARE IMPORTANTĂ: prevenirea “object slicing”.

**Destructori virtuali puri**

***Dacă vrem să eliminăm porțiuni alocate dinamic și pentru clasa derivată dar facem upcasting trebuie să folosim destructori virtuali.***

**Utilizare**: recomandat să fie utilizat dacă mai sunt și alte funcții virtuale. **Obs**. Nu are nici un efect dacă nu se face upcasting.

**Restricție**: trebuiesc definiți în clasă (chiar dacă este abstractă). Dar la moștenire nu mai trebuiesc redefiniti (se construiește un destructor din oficiu), pt a preveni instantierea clasei.

**Downcasting**

* + Folosit in ierarhii polimorfice (cu funcţii virtuale).

**Problema**: upcasting e sigur pentru că respectivele funcţii trebuie să fie definite în bază, downcasting e problematic. => Explicit cast prin: **dynamic\_cast**

***Dacă ştim cu siguranţă tipul obiectului putem folosi “static\_cast”,*** care întoarce pointer către obiectul care satisface cerinţele sau 0 => foloseşte tabelele VTABLE pentru determinarea tipului.

Templates

1. Functii generice

* dacă nu există template<> const char\* -pt "ab" se alege șablonul general
* dacă nu există template<> char\* -pt v1 se alege șablonul general
* /\* NU FACE CONVERSIA NICI (char \*) --> (const char \*) şi nici (const char \*) --> (char \*) \*/

***Overload pe șabloane - ce funcție se apelează (ordinea de alegere)***

pas 1 potrivire FĂRĂ CONVERSIE

* prioritate varianta non-template,
* apoi template fără parametri,
* apoi template cu 1 parametru ,
* apoi template cu mai mulți parametrii

pas 2 dacă nu există potrivire exactă

- conversie DOAR la varianta non-template

1. Clase generice

Funcțiile membru ale unei clase generice sunt şi ele generice (în mod automat).

Nu e necesar să le specificăm cu template daca sunt definite in clasa. este necesar să le specificăm cu template dacă le definim în afara clasei.

**template** <**class** Type1, **class** Type2> **class** myclass {

Type1 i;

Type2 j;

public:

myclass(Type1 a, Type2 b) { i = a; j = b; }

**void** show() { cout << i << ' ' << j << '\n'; }

};

**int** main() {

myclass<**int**, **double**> ob1(10, 0.23);

myclass<**char**, **char** \*> ob2('X', "Templates add power.");

ob1.show(); // show int, double

ob2.show(); // show char, char \*

return 0;

}

**Standard Template Library (STL)**

= bibliotecă de clase C++, parte din Standard Library (std)

**Ofera:**

* structuri de date și algoritmi fundamentali 🡪 dezvoltarea de programe in C++;
* componente generice, parametrizabile. Aproape toate clasele din STL sunt parametrizate (Template).

Componentele STL se pot compune cu usurință fără a sacrifica performanța (generic programming)

STL conține clase pentru:

***◦* containere**

**◦ iteratori**

**◦ algoritmi**

◦ functori (function objects)

◦ allocators

**Container**

**Tipuri de containtere:**

* ***de tip secventa*** (in terminologia STL, o secventa este o lista liniara):
  + vector
  + deque
  + list - implementat ca și listă dublu înlănțuită
* ***asociativi*** (permit regasirea eficienta a informatiilor bazandu-se pe chei)
  + set
  + multiset
  + map (permite accesul la informatii cu cheie unica)
  + multimap
* ***adaptor de containere***
  + stack
  + queue
  + priority\_queue

**List** are performanțe bune la:

▪ Ștergere/adăugare de elemente pe orice poziție (constant time).

▪ Mutarea de elemente sau secvențe de elemente în liste sau chiar și intre liste diferite(constant time).

▪ Iterare de elemente in ordine (linear time)

**Vector** vs **Deque**

* Accesul la elemente de pe orice poziție este mai eficient la vector
* Inserare/ștergerea elementelor de pe orice poziție este mai eficient la Deque (dar nu e timp constant)
* Pentru liste mari Vector alocă zone mari de memorie, deque aloca multe zone mai mici de memorie – Deque este mai eficient în gestiunea memoriei

Exemplu de utilizari:

* Stiva: literele unui cuvant, taskuri (implem de sist de op)
* Coada cu prioritati: taskuri
* Map: (id, date), catalog

**Iterator**

* un concept fundamental in STL, este elementul central pentru algoritmi oferiţi de STL;
* obiect care gestionează o poziție (curentă) din containerul asociat;
* suport pentru traversare (++,--), dereferențiere (\*it);
* permite decuplarea intre algoritmi si containere.

***Tipuri de iteratori:***

• iterator input/output (istream\_iterator, ostream\_iterator)

• forward iterators, iterator bidirectional, iterator random access

• reverse iterators.

Adaptoarele de containere (container adaptors) - stack, queue, priority\_queue - nu oferă iterator**!**

**Lambda Expressions**

int main() {

    vector<int> v={1,2,1,3,1,1};

    for\_each(v.begin(), v.end(), **[ ](int i){ cout << i << ' '; }**);

    cout << '\n';

    int cont = 0;

    for\_each(v.begin(), v.end(),**[&cont](int i){**

**if (i == 1)**

**cont++;**

**}**);

    cout<< cont <<" de 1 ";

}

**Principiile proiectarii de clase (S.O.L.I.D)**

[**Single-responsibility principle**](https://en.wikipedia.org/wiki/Single-responsibility_principle)

A [class](https://en.wikipedia.org/wiki/Class_(computer_programming)) should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.

[**Open–closed principle**](https://en.wikipedia.org/wiki/Open%E2%80%93closed_principle)

"Software entities ... should be open for extension, but closed for modification."

“deschis la extensii” = comportarea modulului poate fi extinsa pentru a satisface noile cerinte.

“inchis la modificare” = nu este permisa modificarea codului sursa.

[**Liskov substitution principle**](https://en.wikipedia.org/wiki/Liskov_substitution_principle)

"Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program."

[**Interface segregation principle**](https://en.wikipedia.org/wiki/Interface_segregation_principle)

"Many client-specific interfaces are better than one general-purpose interface.”

[**Dependency inversion principle**](https://en.wikipedia.org/wiki/Dependency_inversion_principle)

One should "depend upon abstractions, [not] concretions.”

**Design Patterns**

Un **sablon de proiectare** descrie

* o problema care se intalneste in mod repetat in proiectarea programelor
* solutia generala pentru problema respectiva, folosind **clase** si **obiecte**

Atat descrierea problemei cat si a solutiei sunt abstracte astfel incat sa poata fi folosite in multe situatii diferite.

***Abstract Server***

Cand un client depinde direct de server, este incalcat principiul de inversare a dependentelor. Modificarile facute in server se vor propaga in client, iar clientul nu va fi capabil sa foloseasca alte servere similare cu acela pentru care a fost construit. Situatia se poate imbunatati prin inserarea unei interfete abstracte intre client si server

Client -> Interfata <- Manager

***Adapter***

Cand inserarea unei interfete abstracte nu este posibila deoarece serverul este produs de o alta companie (third party ISV) sau are foarte multe dependente de intrare care-l fac greu de modificat, se poate folosi un ADAPTER pentru a lega interfata abstracta de server.

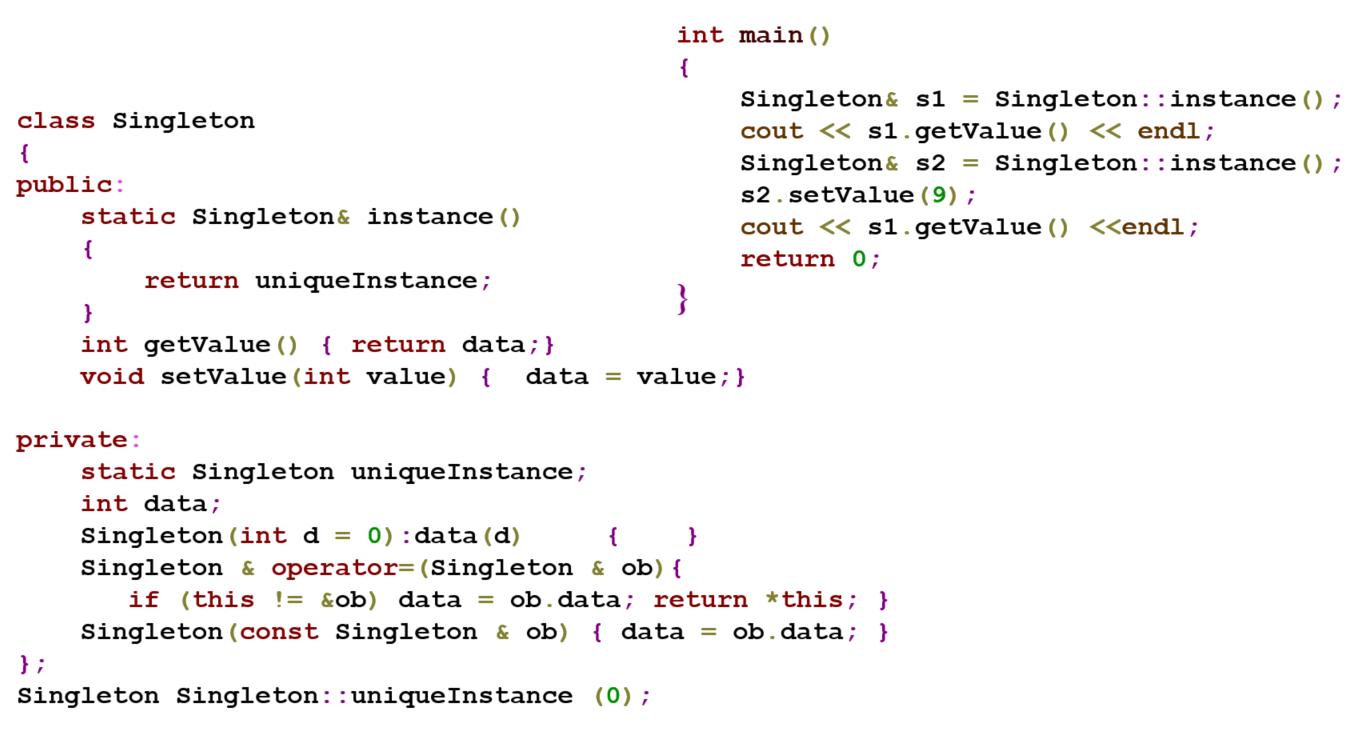
Client -> Interfata <- Adapter ->Manager.

***Singleton (clase cu o singura instanta)***

Intentia: proiectarea unei clase cu un singur obiect (o singura instanţă)

Motivatia: într-un sistem de operare: exista un sistem de fişiere si un singur manager de ferestre

Aplicabilitate**:** când trebuie să existe exact o instanta: clientii clasei trebuie sa aiba acces la instanta din orice punct bine definit.



***Observer***

Intentia: Defineste o dependenta “unul la mai multi” intre obiecte, astfel incat atunci cand unul dintre obiecte isi schimba starea toate obiectele dependente sunt notificate si actualizate automat.  
Alte nume prin care este cunoscut: Dependents, Publish-Subscribe.  
Motivatia: descrie cum sa se stabileasca relatiile intre clase.

* + Un subiect poate avea orice numar de observatori dependenti.
  + Toti observatorii sunt notificati ori de cate ori subiectul isi schimba starea.

Ca raspuns la notificare, fiecare observator va interoga subiectul pentru a-si sincroniza starea cu starea subiectului.

***Aplicabilitatea:***

- cand o abstractie are doua aspecte, unul dependent de celalalt (daca sunt incapsulate in obiecte separate, ele pot fi reutilizate independent).

- cand modificarea unui obiect necesita modificarea altor obiecte si nu se stie cate obiecte trebuie sa fie modificate.

- cand un obiect trebuie sa notifice alte obiecte fara a face presupuneri despre cine sunt aceste obiecte.

***Abstract Object Factory***

**intentie:**

• de a furniza o interfata pentru crearea unei familii de obiecte intercorelate sau dependente fara a specifica clasa lor concreta.

**aplicabilitate:**

• un sistem ar trebui sa fie independent de modul in care sunt create produsele, compuse sau reprezentate

• un sistem ar urma sa fie configurat cu familii multiple de produse

• o familie de obiecte intercorelate este proiectata pentru astfel ca obiectele sa fie utilizate impreuna

• vrei sa furniziei o biblioteca de produse ai vrei sa accesibila numai interfata, nu si implementarea.

**implementare**

* o functie delegat (callback) este o functie care nu este invocata explicit de programator; responsabilitatea apelarii este delegata altei functii care primeste ca parametru adresa functiei delegat

Fabrica de obiecte utilizeaza functii delegat pentru crearea de obiecte: pentru fiecare tip este delegata functia carea creeaza obiecte de acel tip.