* Clasa simpla:

class Ceva{ //s-a declarat o clasa cu numele Ceva, cuvantul class trebuie sa fie scris cu minuscule

}; //atentie, in C++ trebuie sa scriem punct si virgula (;) dupa acolada inchisa (})

* Clasa cu cei trei specificatori de acces / etichete: public, private, protected

Aceasta contine:

-atribute, cu alte cuvinte, atributele sunt variabile

-metode, cu alte cuvinte , metodele sunt functii

* Biblioteca iostream (de la input output stream) este o colectie de programe care ne ajuta la operatiile de citire de la tastatura si scriere pe ecran.

Liniile ce încep cu simbolul # prezintă directive pentru compilator. În acest caz se comunică

compilatorului faptul că trebuie avută în vedere la compilarea si execuția programului biblioteca ce

conține funcțiile de citire/afi܈sare (input/output) din C++.

* Un namespace este un dictionar care ne ajuta sa identificam si sa folosim diferite componente ale unui program using namespace std indica compilatorului ca dorim sa folosim namespace-ul standard pentru C++

#include<iostream> //am introdus biblioteca iostream

using namespace std; //

class Carte { //am creat clasa cu numele Carte

private: //specificator de acces / eticheta private

float pret; //am declarat atributul pret de tip float

int nr\_pagini; //am declarant atributul nr\_pagini de tip int

char autor[64], editura[64];//am declarant 2 atribute de tip char

protected: //specificator de acces / eticheta protected

public: //specificator de acces / eticheta public

void afisare\_Mesaj(); //am declarant o functie voi cu numele afisare\_Mesaj

};

void Carte::afisare\_Mesaj() { // :: este operator de rezolutie global,am definit

//functia declarata in clasa

cout << "Aceasta este o functie care doar afiseaza acest mesaj." << endl;

} //mai sus s-a scris extern functia care a fost declarata in clasa si aceasta doar

//afiseaza un mesaj

int main() { //acesta este corpul principal al programului, putea fi scris si void

//main() dar nu mai era necesar return 0

Carte prima; //am declarat un obiect, mai exact un obiect este o variabila de //tipul unei clase, spre exemplu, daca avem int prim, int e tipul

// si prim e variabila, in cazul nostru Carte este tipul (este o

// clasa) si prima este variabila, mai exact obiectul

//ATENTIE!!!! Pentru a accesa functiile membre (metodele) si/sau atributele, acestea se pot accesa doar

//din public

//Daca avem o metoda sau attribute private, trebuie sa creem functii sau atribute in public care sa le acceseze pe cele din private

prima.afisare\_Mesaj();//am utiliza operatorul punct (.) pentru a

//invoca functii membre ale clasei

return 0; //am scris return 0 pentru ca programul principal este o functie

//int, adica trebuie returnat ceva

}

Alte specificatii:

* Daca nu scriem niciun specificator de acces , cel default este private

Voi crea alt program in care clasa sa contina functii si atribute private ce vor fi accesate prin public:

#include<iostream>

#include<string.h>

using namespace std;

class Carte {

private:

int nr\_pagini;

float pret;

char autor[64], editura[64];

void afisare\_pret();

void afisare\_nr\_pagini();

void afisare\_autor\_editura();

protected:

public:

void atribuie\_pret(float prt);

void atribuie\_nr\_pagini(int numar);

void atribuire\_autor\_editura(const char\* aut,const char\* edit);

void afisare\_Carte();

};

void Carte::atribuie\_pret(float prt) {

pret = prt;

}

void Carte::afisare\_pret() {

cout << "Pretul acestei carti este de " << pret << " lei." << endl;

}

void Carte::atribuie\_nr\_pagini(int numar) {

nr\_pagini = numar;

}

void Carte::afisare\_nr\_pagini() {

cout << "Cartea are " << nr\_pagini << " pagini." << endl;

}

void Carte::atribuire\_autor\_editura(const char\* aut,const char\* edit) {

strcpy(autor, aut);

strcpy(editura, edit);

}

void Carte::afisare\_autor\_editura() {

cout << "Autorul acestei carti este " << autor << " iar editura este " << editura << endl;

}

void Carte::afisare\_Carte() {

afisare\_pret();

afisare\_nr\_pagini();

afisare\_autor\_editura();

}

int main() {

Carte Singur\_pe\_Lume;

Singur\_pe\_Lume.atribuie\_pret(20.5);

Singur\_pe\_Lume.atribuie\_nr\_pagini(415);

Singur\_pe\_Lume.atribuire\_autor\_editura("Hector Mallot", "Teora");

Singur\_pe\_Lume.afisare\_Carte();

}

**Aspecte generale clase**

* Clasele sunt asemanatoare ca si structurile, dar are cateva diferente
* Clasele nu pot sa faca functii repetitive (for, while, do-while,etc)
* Obiectele sunt instante ale clasei, adica sunt niste variabile de tipul clasei
* Clasele au 3 tipuri de etichete/ specificatori de acces : private, protected si public
* Specificatorul de acces default este private.
* Nu putem accesa/invoca metode sau atribute care sunt private, trebuie sa folosim metode pubice pentru a le putea apela pe cele din private.
* In programul principal a invoca functii membre ale clasei avem nevoie de operatorul punct (.)
* Metodele unei clase sunt functii.
* Variabilele declarate in clasa se numesc atribute.
* Setterele sunt metode/functii care seteaza o valoare, iar getterele sunt metode/functii care returneaza ceva.

**Settere si gettere**

Setterele sunt metode/functii dintr-o clasa care seteaza o valoare, iar getterele sunt metode/functii care returneaza ceva.

Exemplu de program cu settere si gettere:

#include <iostream>

using namespace std;

class Data {

private:

int m\_nLuna;

int m\_nZi;

int m\_nAn;

public:

//Ghettere : sunt functii care returneaza ceva, iar numele functiilor pot fi

//oricare

int GetMonth() { return m\_nLuna; }

int GetDay() { return m\_nZi; }

int GetYear() { return m\_nAn; }

//Settere: sunt functii care seteaza variabilele, numele lor pot fi oricare

void SetMonth(int nLuna) { m\_nLuna = nLuna; }

void SetDay(int nZi) { m\_nZi = nZi; }

void SetYear(int nAn) { m\_nAn = nAn; }

};

int main()

{

Data perioada;

perioada.SetDay(01);

perioada.SetMonth(02);

perioada.SetYear(1996);

cout <<"Sunt nascut in data de: ziua "<< perioada.GetDay()<<", luna "<<perioada.GetMonth()<<", anul "<<perioada.GetYear()<<" ."<<endl;

return 0;

}

**Constructori si destructori**

* Un constructor este un membru ce este o functie ce primeste acelasi nume ca si numele clasei, nu este de nici un tip (nici macar void) si nu returneaza nimic.
* În cazul obiectelor, inițializarea la declarare se face prin intermediul unor funcții speciale numite constructori.
* Constructorul are rolul de a initializa variabile si de alocare ale acestora.
* Orice clasa creata caruia nu ii creem un constructor, aceasta isi creeaza singura un constructor gol de forma: nume\_constructor() { }
* In constructor facem initializarile variabilelor (cele din private, o parte sau toate ) si putem chema functii cu aceasta.
* O clasa poate avea mai multi constructori.
* Constructorii se declara in domeniul public al clasei
* Exista 3 tipuri de constructori:
  + Constructor fara parametrii
  + Constructori cu parametrii
  + Constructori de copiere
* Constructorii nu se mostenesc
* Nu pot primi ca parametri instante ale clasei ce se defineste,ci doar pointeri sau referinte la instantele clasei respective
* Constructorii cu parametrii sunt de 2 tipuri:
  + Cu parametri ce NU iau valori implicite
  + Cu parametri ce iau valori implicite

Mai jos voi scrie un program care foloseste un constructor:

#include<iostream>

using namespace std;

class Fractie {

private:

int numitor, numarator;

public:

Fractie(); //am creat constructorul

protected:

};

Fractie::Fractie() { //am definit constructorul extern

numitor = 0;

numarator = 1;

cout << endl << "Hei din constructor gol";

}

int main()

{

Fractie \*f1, f2; //atentie, cand creem obiecte statice, se va apela automat

//constructorul si se va afisa mesajul din constructor

//pentru obiectele cu pointer, alocare dinamica, nu se va //afisa mesajul

//destructorul este apelat cu ajutorul kwarg-ului (sau kword, nu stiu sigur) //delete

delete f1; //se apeleaza delete de un obiect DOAR CAND E ALOCAT DINAMIC

return 0;

}

Voi crea un program care are mai multi constructori:

#include<iostream>

#include<cassert>

#include<cstring>

using namespace std;

class Fractie {

private:

int numitor, numarator;

char\* tip\_fractie;

public:

void afisare()

{

cout << "Tipul fractiei este :" << tip\_fractie << ", numitorul acesteia este: " << numitor << ", iar numaratorul este: " << numarator << endl;

}

Fractie() {

numitor = 1;

numarator = 0;

tip\_fractie = new char[strlen("Necunoscut" + 1)];

strcpy(tip\_fractie, "Necunoscut");

cout << endl << "S-a apelat constructorul fara parametri" << endl;

}

Fractie(const char\* tip, int numa, int numi = 1) {

assert(numi != 0);

this->tip\_fractie = new char[strlen(tip) + 1];

strcpy(this->tip\_fractie, tip);

numarator = numa;

numitor = numi;

cout << endl << "S-a apelat contructorul cu cei trei parametri char int int" << endl;

}

/\*

Fractie(const char\* tip, double numa, int numi = 1) {

cout << endl << "S-a apelat constructorul cu cei trei parametri char double int" << endl;

}\*/

~Fractie() { //Destructor, este unic, 1 pe clasa creata, nu are parametri

delete[]tip\_fractie;

cout << endl << "S-a apelat detructorul" << endl;

}

protected:

};

int main()

{

{

Fractie\* f1, f2, f3("Subunitara", 1, 2)/\*, f4("Supraunitara", 3.5, 2)\*/; //atentie, cand creem obiecte statice, se va apela automat constructorul si se va afisa mesajul din constructor

//pentru obiectele cu pointer, alocare dinamic, nu se va afisa mesajul

//in cazul nostru f1 e cu pointer

f1 = new Fractie(); //pentru a apela constructorul si a-l aloca dinamic folosim new , alocam pe heep cele 2 variabile, cele din constructor ( f2 primeste new Fractie )

//iar in cazul asta se va apela automat mesajul din contructor gol

f2.afisare();

//f4.afisare();

//destructorul este apelat cu ajutorul kwarg-ului (sau kword, nu stiu sigur) delete

delete f1; //se apeleaza delete de un obiect DOAR CAND E ALOCAT DINAMIC

}

cout << endl << "Mesaj return" << endl;

return 0;

}

Sau mai putem scrie in alt mod, prin declarare functii si constructori in clasa si definire in afara acesteia:

#include<iostream>

#include<cassert>

#include<cstring>

using namespace std;

class Fractie {

private:

int numitor, numarator;

char\* tip\_fractie;

public:

void afisare();

Fractie(); //definire constructor implicit

Fractie(const char\* tip, int numa, int numi); //definire constructor cu 3 parametri

/\*

Fractie(const char\* tip, double numa, int numi = 1) {

cout << endl << "S-a apelat constructorul cu cei trei parametri char double int" << endl;

}\*/

~Fractie() { //Destructor, este unic, 1 pe clasa creata, nu are parametri

delete[]tip\_fractie;

cout << endl << "S-a apelat detructorul" << endl;

}

protected:

};

// declarari externe ale metodelor si constructorilor

void Fractie::afisare() //definire metoda/functie afisare

{

cout << "Tipul fractiei este :" << tip\_fractie << ", numitorul acesteia este: " << numitor << ", iar numaratorul este: " << numarator << endl;

}

Fractie::Fractie() { //definire constructor implicit

numitor = 1;

numarator = 0;

tip\_fractie = new char[strlen("Necunoscut" + 1)];

strcpy(tip\_fractie, "Necunoscut");

cout << endl << "S-a apelat constructorul fara parametri" << endl;

}

Fractie::Fractie(const char\* tip, int numa, int numi = 1) { //definire constructor cu parametri

assert(numi != 0);

this->tip\_fractie = new char[strlen(tip) + 1];

strcpy(this->tip\_fractie, tip);

numarator = numa;

numitor = numi;

cout << endl << "S-a apelat contructorul cu cei trei parametri char int int" << endl;

}

int main()

{

{

Fractie\* f1, f2, f3("Subunitara", 1, 2)/\*, f4("Supraunitara", 3.5, 2)\*/; //atentie, cand creem obiecte statice, se va apela automat constructorul si se va afisa mesajul din constructor

//pentru obiectele cu pointer, alocare dinamic, nu se va afisa mesajul

//in cazul nostru f1 e cu pointer

f1 = new Fractie(); //pentru a apela constructorul si a-l aloca dinamic folosim new , alocam pe heep cele 2 variabile, cele din constructor ( f2 primeste new Fractie )

//iar in cazul asta se va apela automat mesajul din contructor gol

f2.afisare();

//f4.afisare();

//destructorul este apelat cu ajutorul kwarg-ului (sau kword, nu stiu sigur) delete

delete f1; //se apeleaza delete de un obiect DOAR CAND E ALOCAT DINAMIC

}

cout << endl << "Mesaj return" << endl;

return 0;

}

Exemplu simplu constructor implicit si constructor cu parametri:

#include<iostream>

#include<cmath>

using namespace std;

class Nr\_complex {

private:

float re, im;

public:

void citire();

void afisare();

float modul();

Nr\_complex(); //Declarare constructor implicit

Nr\_complex(float r, float i); //Declarare constructor cu parametri

protected:

};

// definire metode si constructori

void Nr\_complex::citire() {

cout <<endl<< "Dati partea reala: ";

cin >> re;

cout << endl << "Dati partea imaginara: ";

cin >> im;

}

void Nr\_complex::afisare() {

cout << endl << re << "+" << im<<"\*i";

}

float Nr\_complex::modul() {

return sqrt(re \* re + im \* im);

}

Nr\_complex::Nr\_complex() { //definire constructor implicit

re = 0;

im = 0;

cout << "Apel constructor implicit " << endl;

}

Nr\_complex::Nr\_complex(float r, float i) { //Definire constructor cu parametri

re = r;

im = i;

cout << "Apelare constructor cu 2 parametri" << endl;

}

int main()

{

{

Nr\_complex z1, z2(7,2); //Declarare obiecte/instante

z1.afisare();

z2.afisare();

}

cout << endl << "Mesaj return" << endl;

return 0;

}

Exemplu constructori cu parametri impliciti:

#include<iostream>

#include<cmath>

using namespace std;

class Nr\_complex {

private:

float re, im;

public:

void citire();

void afisare();

float modul();

Nr\_complex(float r, float i); //Declarare constructor cu parametri impliciti

//ATENTIE:in declararea functiilor nu atribuim valori,

//doar in definirea acestora

protected:

};

// declarari externe ale metodelor si constructorilor

void Nr\_complex::citire() {

cout << endl << "Dati partea reala: ";

cin >> re;

cout << endl << "Dati partea imaginara: ";

cin >> im;

}

void Nr\_complex::afisare() {

cout << endl << re << "+" << im << "\*i";

}

float Nr\_complex::modul() {

return sqrt(re \* re + im \* im);

}

Nr\_complex::Nr\_complex(float r = 0, float i = 0) { //definire constructor cu parametri

//impliciti

im = i;

re = r;

cout << "Apelare constructor cu 2 parametri" << endl;

}

int main()

{

{

Nr\_complex z1(2, 3), z2(4), z3 = 5, z4; //declarare instante/obiecte

Nr\_complex z5 = z4; // putem declara si initializa un obiect si in modul asta

z1.afisare(); //pentru z1(2,3) se va afisa 2+3\*i

z2.afisare(); //pentru z2(4) se va afisa 4+0\*i

z3.afisare(); //pentru z3= 5 se va afisa 5+0\*i

z4.afisare(); //pentru z4 se va afisa 0+0\*i

}

cout << endl << "Mesaj return" << endl;

return 0;

}

**Constructori de copiere:**

* Initializarea obiectelor la declarare cu alte obiecte deja create

Constructori de copiere - Utilizare

Constructorii de copiere se apelează în următoarele cazuri:

* Crearea de obiecte cu iniţializare, pornind de la un obiect care există (IdClasa ob2=ob1 sau IdClasa ob2(ob1)).
* Apelul unei funcţii care lucrează cu obiecte transferate prin valoare, când este nevoie de crearea unei copii a obiectului pe stivă (cazul f(ob);).
* Returnarea dintr-o funcţie a unui obiect prin valoare (return ob;)

Exemplu constructori de copiere:

#include<iostream>

#include<cmath>

using namespace std;

class Nr\_complex {

private:

float re, im;

public:

void citire();

void afisare();

float modul();

Nr\_complex(float r, float i); //Declarare constructor cu parametri impliciti

//ATENTIE:in declararea functiilor nu atribuim valori,

//doar in definirea acestora

Nr\_complex(const Nr\_complex& z);

protected:

};

// declarari externe ale metodelor si constructorilor

void Nr\_complex::citire() {

cout << endl << "Dati partea reala: ";

cin >> re;

cout << endl << "Dati partea imaginara: ";

cin >> im;

}

void Nr\_complex::afisare() {

cout << endl << re << "+" << im << "\*i";

}

float Nr\_complex::modul() {

return sqrt(re \* re + im \* im);

}

Nr\_complex::Nr\_complex(float r = 0, float i = 0) { //definire constructor cu parametri

//impliciti

im = i;

re = r;

cout << "Apelare constructor cu 2 parametri" << endl;

}

Nr\_complex::Nr\_complex(const Nr\_complex& z) {

re = z.re;

im = z.im;

cout << "Apel constructor de copiere." << endl;

}

void f(Nr\_complex) {

cout << "Mesaj din functia void f" << endl;

}

int main()

{

{

Nr\_complex z1(2, 3); //declarare instante/obiecte

Nr\_complex z2 = z1;

z1.afisare(); //pentru z1(2,3) se va afisa 2+3\*i

z2.afisare(); //pentru z2(4) se va afisa 4+0\*i

cout << endl;

f(z1); //apel constructor de copiere

}

cout << endl << "Mesaj return" << endl;

return 0;

}

**Destructorul** este o functie membra speciala a unei clase ce apeleaza in mod automat distrugerea unui obiect.

Are rolul de a elibera zonele alocate dinamic, a resurselor, etc.

**Alt tip de tablou de obiect/vector de obiect:**

#include <iostream>

#include<cstring>

using namespace std;

class File {

private:

//char\* path;

int mode, a;

public:

File(int aa, int open\_mode);

};

File::File(int aa = 1, int open\_mode = 1) {

cout << "Mesaj Constructor si valorile acestuia" << endl;

cout << aa << " " << open\_mode << endl;

}

using namespace std;

int main() {

File folder[2] = { File(1,1),File(2,2) }; //obiectul folder[2] contine 2 obiecte,

//este un tablou de obiecte

cout << "S-a executat return 0";

return 0;

}

**Conversie implicita:**

#include <iostream>

#include<cstring>

using namespace std;

class sir {

int size;

int capacity;

char\* buffer;

public:

sir();

sir(int size);

sir(const char\*);

};

sir::sir() {

cout << "Mesaj constructor default" << endl;

}

sir::sir(int size) {

cout << "Mesaj constructor cu parametru int size" << endl;

}

sir::sir(const char\*) {

cout << "Mesaj constructor cu parametru const\* char" << endl;

}

int main() {

sir s = "Salut!";

s = 1; //compilatorul interpreteaza prin s = sir(1), apeland constructorul

//cu argument

//se numeste conversie implicita

cout << "S-a executat return 0";

return 0;

}

**Constructor explicit:**

#include <iostream>

#include<cstring>

using namespace std;

class sir {

int size;

int capacity;

char\* buffer;

public:

sir();

explicit sir(int size);

sir(const char\* ); //pot pune si variabila dupa \*

};

sir::sir() {

cout << "Mesaj constructor default" << endl;

}

sir::sir(int size) {

cout << "Mesaj constructor cu parametru int size" << endl;

}

sir::sir(const char\*) { //la fel si aici, pot pune variabila de sus

cout << "Mesaj constructor cu parametru const\* char" << endl;

}

int main() {

sir s = "Salut!";

// s = 1; //compilatorul interpreteaza prin s = sir(1), apeland constructorul cu

//argument

//se numeste conversie implicita

//va da eroare la compilare datorita constructorului care este explicit si //daca dorim conversia va trebui sa scriem:

s = (sir)1;

//sau

s = sir(1);

cout << "S-a executat return 0";

return 0;

}