Facultatea de Matematică și Informatică,
Universitatea din București

Tutoriat 1

Programare Orientată pe Obiecte - Informatică, Anul I

Tudor-Gabriel Miu
Radu Tudor Vrînceanu
3-15-2024

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Cuprins

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Introducere în limbaj

Tipuri de fișiere în C++ (.cpp, .h)

Limbajul de programare C++ permite stocarea mai multor tipuri de fișiere, printre care fișiere de tip header (.h) și fișiere de tip sursă (.cpp). Cele dintâi permit declararea structurii (antetelor) pentru anumite funcții, clase, structuri de date, etc…, cele din urmă fiind utilizate pentru implementarea acestora.

Cum arată un program în C++

```
// define-uri proprii i.e.
#define daca if // sunt niste alias-uri pentru anumite lucruri deja definite
de catre noi sau limbaj (NU SUNT RECOMANDATE!)

// include-uri de librarii
#include <librariax> // daca libraria de baza este in C++
#include <clibrarie> // se pune un c in fata daca libraria a fost nativa in C
i.e. math.h din C devine cmath in C++

// diverse namespace-uri -> colectii de functii care sunt inregistrate sub un
scop
using namespace x;

int main() {
    // instructiuni
    return 0;
}
```

Tipuri de date în C++

Limbajul C++ este unul puternic tipizat (strongly typed), prezentând astfel câteva tipuri de bază pe care le puteți găsi aici (https://cplusplus.com/doc/tutorial/variables) cu tot cu limita superioară datorată stocării lor pe un număr finit de octeți. Dat acest fapt, aveți mare grijă la **overflow-uri** pentru valorile variabilelor, în special numerice. Amintiți-vă de la Arhitectura Sistemelor de Calcul faptul că un int este reprezentat pe 4 octeți (32 de biți) iar MSB (most significant bit) este bitul de semn, dacă încercăm să stocăm numere mai mari strict decât 2^31 - 1 într-un int acestea vor deveni negative datorită populării cu 1 a bitului de semn.

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Pointeri în C++

Un pointer în limbajul de programare C++ este o variabilă care stochează adresa de memorie a altei variabile. În esență, un pointer "pointează" către locația în memorie a unei alte variabile. Utilizarea pointerilor ne oferă capacitatea de a manipula direct datele în memorie și de a accesa sau modifica valorile acestora.

Să examinăm o declarație de pointer simplă:

```
int *p;
```

În această declarație, `p` este un pointer către un integer (`int`). Putem atribui adresa de memorie a unei variabile de tip integer către acest pointer, astfel:

```
int x = 32;

int *p = x; // eroare la compilare deoarece x nu este adresa unei variabile, ci reține valoarea 32

int *p = &x; // & este operatorul pe care îl folosim pentru a prelua adresa unei variabile când acesta stă lângă una, vom vedea că există și un alt caz pentru acest operator. Astfel în x vom reține o adresă de memorie unde a fost declarat x-ul nostru (precum 0xFFBEE2320).

cout << p << ' ' << *p << endl; // ce va afișa aici? p va afișa adresa de memorie unde a fost declarat x-ul iar *p va afișa VALOAREA DIN ADRESA DE MEMORIE CĂTRE CARE POINTEAZĂ (ARATĂ), în cazul nostru p reține adresa lui x, deci *p afișează valoarea din adresa de memorie unde a fost declarat x.
```

Având astfel o scurtă introducere asupra pointerilor, vrem să vedem niște cazuri despre ce se întâmplă în anumite cazuri de funcționalitate:

```
int x = 32;
int *p = &x;

*p = 54;

cout << x << endl; // ce se afișează aici, 54 sau 32? de ce? R: 54, pentru că noi schimbăm VALOAREA DIN LOCAȚIA DE MEMORIE CĂTRE CARE ARATĂ p, iar p arată către zona de memorie unde a fost definit x, deci practic noi schimbăm valoarea lui x ÎN ACEST CAZ!
```

Alocarea dinamică a memoriei și gestionarea Pointerilor

Alocarea dinamică a memoriei ne permite să rezervăm spațiu în memoria de la nivelul sistemului de operare în timpul rulării programului. Aceasta este utilă atunci

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

când nu știm dimensiunea exactă a datelor pe care le vom utiliza în avans sau când dorim să creăm și să distrugem obiecte în timpul execuției.

În C++, putem aloca și dezaloca dinamic memorie folosind operatorii new și delete. De exemplu:

```
int *p = new int(32); // reține un pointer către o adresă de memorie unde se stochează valoarea 32.

cout << p << ' ' << *p << endl; // afișează adresa pe care o reține p și valoarea din ZONA DE MEMORIE CĂTRE CARE POINTEAZĂ p

if (p != NULL) // verificăm dacă pointerul nu este null
delete p; // îl ștergem din zona de memorie pentru a evita MEMORY LEAKAGE
```

Alocarea dinamică de date în memorie implică o atenție puțin mai mare pentru gestionarea memoriei și anume la scurgeri de memorie.

Haideți să vedem ce se întâmplă în momentul în care avem array-uri alocate dinamic, vom folosi cazul n = 1 (vector):

```
int *p = new int[1024];
p[0] = 32;

cout << p << ' ' << p[0] << endl; // afișează adresa de memorie a primului element din array și valoarea din prima căsuță de memorie

if (p)
delete p; // șterge p, dar este corect? R: nu el va șterge doar prima adresa de memorie. iar restul array-ului va rămâne pierdut în memorie.
delete[] p; // șterge tot array-ul din memorie.
```

Este important să fim atenți la gestionarea corectă a memoriei atunci când lucrăm cu alocarea dinamică. Este responsabilitatea noastră să eliberăm memoria pe care am alocat-o dinamic, pentru a preveni pierderea de memorie și alte probleme legate de gestionarea resurselor.

Ce se intampla dacă nu eliberăm memoria alocată dinamic?

Referința în C++

Referințele în C++ sunt aliasuri pentru alte obiecte. O referință oferă un alt nume pentru o variabilă deja existentă, ceea ce permite să lucrăm cu acea variabilă sub un alt nume. Deși referințele sunt similare cu pointerii în unele privințe, ele sunt gestionate în mod diferit și oferă o sintaxă mai simplă și mai sigură.

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```
int x = 32;
int& y= x;

int& y = *x; // eroare la compilare x nu este de tip int* (pointer la int) deci nu am cum să-i extrag valoarea

int &y = &x; // eroare la compilare, x chiar dacă este de tip int și &x, înseamnă adresa lui x din memorie am spus ca tipurile int& lucrează exclusiv pe VALORI DEJA DEFINITE ÎN MEMORIE, NU cu adrese.

cout << y << endl; // va afișa 32
```

Acum vrem să vedem ce se întâmplă când un astfel de tip este folosit și ce logică are în spate:

```
int x = 32;
int &ref = x;

ref = 54;

cout << x << endl; // ce valoare se va afișa, 32 sau 54? R: 54, tocmai ce am zis că tipul referință este un alias, o etichetă e ca și cum am scrie x = 54;, dar concret ref este o referință către valoarea din zona de memorie unde a fost declarat x.
```

Acum, `ref` este o referință către variabila `x`. Orice modificare a lui `ref` va afecta variabila `x` și viceversa.

Referințele sunt folosite adesea pentru a evita copierea inutilă a datelor și pentru a lucra cu obiecte existente într-un mod mai eficient și mai clar.

Diferențe între Pointeri și Referințe

- Atribuirea inițială: Un pointer necesită o atribuire inițială a adresei de memorie, în timp ce o referință trebuie inițializată cu o variabilă existentă.

- Reasignarea: Un pointer poate fi reasignat pentru a pointa către alte adrese de memorie, în timp ce o referință nu poate fi reasignată după ce a fost inițializată.

- Verificarea null: Un pointer poate fi verificat pentru a vedea dacă este nul, ceea ce înseamnă că nu pointează către nicio adresă de memorie, în timp ce o referință nu poate fi null și trebuie să fie inițializată cu o variabilă existentă.

- Sintaxa: Operatorul de referință (`&`) este utilizat pentru a crea o referință, în timp ce operatorul de dereferențiere (`*`) este utilizat pentru a accesa valoarea la care pointează un pointer.

- Verificare pentru validitate: Referințele sunt întotdeauna garantate să fie valide, deoarece trebuie să fie inițializate cu o variabilă existentă, în timp ce

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

un pointer ar putea să nu fie valid, ceea ce ar putea duce la comportament nedefinit sau la erori în timpul rulării programului.

În general, referințele sunt preferate atunci când lucrăm cu obiecte existente și când nu avem nevoie de comportamentul suplimentar oferit de pointeri, cum ar fi reasignarea și verificarea pentru null.

Sa vedem niște exemple mai “ciudate”:

```
#include <iostream>
using namespace std;
int main() {

   int i = 10, x=30;
   int &j = i;
   int *pi = &i;

   *pi = 20; //valoarea din i devine 20

   j = x; //valoarea din i devine 30
   j++; //valoarea din i devine 31

   cout << "i = " << i << endl;
   cout << "*pi = " << *pi << endl;
   return 0;
}
```

```
#include <iostream>

using namespace std;

int main() {
   int i = 10, x=30;
   int &j = i;
   int *pi = &i;

   cout << "*pi = " << *pi << endl;

   *pi = *pi + 1; //incrementeaza valoarea din i
   cout << "*pi = " << *pi << endl;

   ++*pi; //merge cum v-ati astepta, incrementeaza valoarea
   cout << "*pi = " << *pi << endl;


   *pi++; //nu merge :) de ce?
   cout << "*pi = " << *pi << endl;

}
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```
#include <iostream>

int main() {

   int i = 10;
   int *pi = &i;
   int x = 420;

   pi += 1; //crestem adresa. Intrucat x e declarat imediat dupa, pi
pointeaza catre el. Daca nu era nimic declarat, arata spre ceva random din
memorie

   std::cout << "i = " << i << std::endl; //afisam 10
   std::cout << "*pi = " << *pi << std::endl; //afisam 420
   //*pi = *pi * 10;
   //std::cout << x; //afiseaza 4200 :)
   return 0;
}
//daca am decomenta liniile 15 si 16 am observa ca prin modificarea valorii
spre care pointeaza pi acum se modifica x-ul.
```

```
#include <iostream>

int main() {
   int n = 1;
   int *p = &n;
   int **pp = &p;

   **pp = 3;
   *p = 2;
   n = 1;
   std::cout << n << *p << **pp << std::endl;
}
```

```
#include <iostream>

int main() {
   int n = 1, q = 2;
   int *p = &n;
   int *pp = &q;

   *pp = *p;
   pp = p;

   std::cout << *p << std::endl;
   std::cout << *pp << std::endl;
}
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Calificatorul CONST

Calificatorul `const` în C++ este folosit pentru a declara o constantă sau pentru a specifica că un obiect nu poate fi modificat. Atunci când un obiect este declarat ca `const`, acesta nu poate fi modificat după ce a fost inițializat (ceea ce nu e chiar adevărat, vom vedea pe parcurs metode de a ocoli acest lucru prin care putem modifica variable `const`)

```
const int x = 10; // x este o constantă și nu poate fi modificată
```

Utilizarea `const` este utilă pentru a face codul mai sigur și mai ușor de înțeles, deoarece oferă o garanție că obiectele nu vor fi modificate în mod accidental.

În cazul pointerilor, `const` poate fi folosit pentru a specifica că obiectul către care pointează pointerul este o constantă, sau că pointerul în sine este o constantă, sau ambele.

```
int x = 10;
const int *p1 = &x; // pointer către o constantă
int * const p2 = &x; // pointer constant către o variabilă
const int * const p3 = &x; // pointer constant către o constantă
```

În cazul lui `p1`, pointerul indică către o variabilă care este o constantă și nu poate fi modificată prin intermediul pointerului. În cazul lui `p2`, pointerul în sine este o constantă și nu poate fi reasignat să pointeze către altă adresă de memorie. În cazul lui `p3`, ambele, pointerul și variabila către care pointează, sunt constante.

Ce este Supraincarcarea Funcțiilor?

Supraincarcarea funcțiilor este o caracteristică a limbajului de programare C++ care permite definirea mai multor funcții cu același nume, dar cu semnături diferite. Semnătura unei funcții include numele acesteia împreună cu tipurile și ordinea parametrilor săi. Atunci când apelăm o funcție supraîncărcată, compilatorul selectează versiunea corespunzătoare a funcției pe baza tipurilor și numărului de argumente furnizate. Două funcții nu pot să difere doar prin tipul de date returnat (e evident că, date fiind datele de intrare, compilatorul le poate folosi pe ambele la fel de ușor)

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Demonstrarea Polimorfismului la Compilare

Polimorfismul la compilare se referă la capacitatea unui program de a selecta în mod dinamic o implementare specifică a unei funcții în funcție de tipul sau numărul argumentelor furnizate la apelul funcției. Acest lucru este posibil datorită supraincarcării funcțiilor și a faptului că compilatorul poate rezolva funcțiile supraincarcate la timpul compilării.

În viitor vom vorbi și despre polimorfism la momentul rulării.

Exemple **(așa DA)**:

```cpp
#include <iostream>

int f(int x) {
   return x;
}

int f(int x, int y) {
   return x * y;
}

int main() {
   std::cout << "Hello, World!" << std::endl;
   return 0;
}
```

Exemple **(așa NU)**:

```cpp
#include <iostream>

void f(int x){

}

void f(int x){ //signatura identica cu anterioara

}

int main() {
   int n;
}
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```
#include <iostream>

void f(int *x){

}

void f(int x[]){ //array-ul e, de fapu, pointer. Signatura identica

}

int main() {
   int n;
}
```

Următorul exemplu nu e identificat la compilare, dar dacă decomentăm linia 13 (`f(n);`) se identifică ambiguitatea între cele două declarări.

```
#include <iostream>

void f(int x){

}

void f(int &x){ //signatura nu e chiar identica, dar orice apel se potriveste
cu ambele => ambiguitate
}

int main() {
   int n;
   //f(n);
}
```

Definirea și Utilizarea Funcțiilor cu Valori Implicite

Funcțiile cu valori implicite sunt funcții care au parametri opționali, pentru care compilatorul furnizează automat o valoare implicită dacă nu este furnizată nicio valoare atunci când se apelează funcția. Această caracteristică este utilă pentru a face codul mai concis și mai ușor de utilizat în cazurile în care unele argumente sunt folosite mai des decât altele.

Să vedem un exemplu simplu de funcție cu valori implicite:

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```cpp
#include <iostream>

void salut(const char* nume = "anonim") {
   std::cout << "Salut, " << nume << "!" << std::endl;
}

int main() {
   salut();           // Va afișa "Salut, anonim!"
   salut("Alex");     // Va afișa "Salut, Alex!"

   return 0;
}
```

Valorile implicite se adaugă **DE LA DREAPTA LA STÂNGA**!

(așa NU):

```cpp
void f(int x, int y = 1, int z){
  //acest exemplu nu compilează
}
```

Acum să vedem niște exemple mai ciudate:

Acest exemplu nu funcționează, deoarece apelul `f(n);` se potrivește cu ambele signaturi (ambiguitate)

```cpp
#include <iostream>

void f(int x , int y = 1){

}

void f(int x){

}

int main() {
   int n;
   f(n);
}
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Următorul exemplu funcționează dacă apelăm cu 0 sau 2 valori (se apelează prima funcție). Cu un singur parametru ne lovim de ambiguitatea de mai sus.

```
#include <iostream>

void f(int x = 0 , int y = 1){

}

void f(int x){

}

int main() {
   int n;
   f(n);
}
```

Funcții cu Parametrii const, Funcții care Returnează const, Funcții const

În C++, `const` poate fi utilizat în diverse moduri în definiția funcțiilor:

Funcții cu Parametrii const:

Acestea sunt funcții care primesc unul sau mai mulți parametrii `const`. Acest lucru înseamnă că parametrii nu pot fi modificați în cadrul funcției.

```
void afisare(const int x) {
   std::cout << x << std::endl;
}
```

Funcții care Returnează const:

Acestea sunt funcții care returnează o referință `const`. Acest lucru înseamnă că rezultatul returnat nu poate fi modificat de către apelant.

```
const int& getNumber() {
   return someNumber;
}
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Funcții const:

Acestea sunt funcții membru ale claselor care sunt declarate ca `const`. Aceasta indică faptul că funcția nu modifică starea obiectului pentru care este apelată.

```
class MyClass {
public:
    void print() const {
        std::cout << "Printing..." << std::endl;
    }
};
```

Un exemplu mai complex:

E totul bine?

```
#include <iostream>

int main() {
    const int x = 5;
    int y = 5, z = 5, t = 5;

    const int *ptr0 = &x;
    const int *ptr = &y;
    int const *ptr2 = &z;
    int *const ptr3 = &t;

    y = 6;
    z = 6;
    *ptr3 = 6;

    std::cout << "x: " << x << std::endl;
    std::cout << "y: " << y << std::endl;
    std::cout << "z: " << z << std::endl;
    std::cout << "t: " << t << std::endl;
    std::cout << "*ptr0: " << *ptr0 << std::endl;
    std::cout << "*ptr: " << *ptr << std::endl;
    std::cout << "*ptr2: " << *ptr2 << std::endl;
    std::cout << "*ptr3: " << *ptr3 << std::endl;
}
```

Surprinzător, da, dar compilatorul va da două warninguri:

```
Local variable 'ptr' may point to invalidated memory
Local variable 'ptr2' may point to invalidated memory
```

De ce?

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Dar aici?

```
#include <iostream>

int main() {
   const int x = 5;
   int *ptr = &x;
}
```

Aici nu compilează. Dacă ne-ar permite să facem asta, am putea modifica valoarea lui `x` prin `ptr` mai târziu. Dacă facem și pointerul constant, totul e bine.

Ce se intampla daca o functie primeste ca parametru de tip `const`?
Ce se intampla daca o functie returneaza un `const`?

```
#include <iostream>

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

const int g(int &y) {
   return y;
}

int main() {
   int x = 5;
   int y = f(x);
   x = g(x);
   x++;
   y++;
   std::cout << x << std::endl;
   std::cout << y << std::endl;

}
```

R: Funcția `f` nu va putea modifica valoarea lui `x` în interiorul ei. Funcția `g` returnează un `const`, dar acesta este copiat și transmis înapoi către `x` în `main`, unde nu e `const` și îl putem modifica :).

Transmiterea Parametrilor către Funcții

În C++, parametrii pot fi transmiși către funcții în diverse moduri, inclusiv prin valoare, prin referință și prin pointeri. Fiecare mod de transmitere a parametrilor are avantaje și dezavantaje și este potrivit pentru diferite situații.

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Transmiterea prin Valoare

Atunci când parametrii sunt transmiși prin valoare, o copie a valorii parametrului este creată în interiorul funcției. Orice modificare a parametrilor în interiorul funcției nu afectează parametrii reali din apelul funcției.

```
void func(int x) {
   x = x * 2; // Modificarea parametrului local x
}

int main() {
   int a = 5;
   func(a); // a rămâne 5 în afara funcției
   return 0;
}
```

Transmiterea prin Referință

Transmiterea prin referință implică transmiterea adresei de memorie a parametrilor către funcție. Aceasta permite funcției să lucreze direct cu variabilele originale din apelul funcției și să le modifice.

```
void func(int &x) {
   x = x * 2; // Modificarea parametrului real a din apelul funcției
}

int main() {
   int a = 5;
   func(a); // a devine 10
   return 0;
}
```

Transmiterea prin Pointeri

Transmiterea prin pointeri implică transmiterea adresei de memorie a parametrilor prin intermediul pointerilor către funcție. Aceasta oferă similaritate cu transmiterea prin referință, dar necesită utilizarea explicită a dereferențierii.

```
void func(int *ptr) {
   (*ptr) = (*ptr) * 2; // Modificarea parametrului real a din apelul
funcției
}

int main() {
   int a = 5;
   func(&a); // a devine 10
}
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Funcții în Struct

În C++, putem defini funcții în interiorul structurilor, ceea ce ne permite să asociem funcționalități cu datele într-un mod organizat și encapsulat. Aceste funcții sunt numite funcții membru ale structurii și au acces la toate membrii structurii, inclusiv la membrii privați.

```cpp
#include <iostream>


struct A {
   int x;

   [[nodiscard]] int f(int a) const {
       return a + x;
   }
};


int main() {
   int x = 1;
   A a;
   a.x = 2;
   std::cout << a.f(x) << std::endl;
}
```

Introducere în Programarea Orientată pe Obiecte (POO)

Programarea Orientată pe Obiecte este o paradigmă de programare care se bazează pe conceptul de "obiecte". Un obiect este o entitate care combină date și comportament asociat (metode) într-o singură unitate. Această abordare permite dezvoltatorilor să modeleze și să simuleze problemele din lumea reală într-o manieră mai intuitivă și mai modulară.

Importanța POO în Dezvoltarea Software-ului

POO aduce mai multe beneficii în dezvoltarea software-ului, printre care:

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

- Reutilizarea codului: Odată ce sunt definite și implementate clase și obiecte, acestea pot fi utilizate în diferite părți ale programului, reducând astfel redundanța și creșterea eficienței dezvoltării.
- Încapsularea și Abstracția: POO permite încapsularea datelor și funcționalităților asociate în interiorul obiectelor, permițând astfel dezvoltatorilor să izoleze și să protejeze datele și comportamentele obiectelor de influențe externe, ceea ce duce la un cod mai curat și mai ușor de întreținut.

- Modularitatea: POO promovează împărțirea codului în module și componente independente, ceea ce face mai ușoară dezvoltarea, testarea și întreținerea programelor mari și complexe.

- Extensibilitatea și Flexibilitatea: Prin intermediul conceptelor cum ar fi moștenirea și polimorfismul, POO facilitează extinderea și modificarea funcționalității programului fără a afecta codul existent.

POO vs alte paradigme de programare

În contrast cu alte paradigme de programare, cum ar fi programarea procedurală și programarea funcțională, POO se concentrează pe abordarea problemelor într-un mod mai intuitiv și mai natural, reflectând mai bine modul în care gândim și interacționăm cu lumea reală. În timp ce programarea procedurală se axează pe secvențialitatea instrucțiunilor și pe manipularea datelor prin intermediul funcțiilor, iar programarea funcțională se concentrează pe evaluarea expresiilor și pe utilizarea funcțiilor pure, POO oferă un cadru mai organizat și mai ușor de înțeles pentru dezvoltarea software-ului.

Ce este un Obiect?

În POO, un obiect este o instanță a unei clase. O clasă este un șablon sau un tip de date definit de programator care conține date și funcții care operează asupra acestor date. Un obiect este o entitate independentă care combină datele și comportamentul asociat într-o singură unitate.

Folosiți getteri și setteri! În exemplele următoare am utilizat câmpuri publice în interiorul claselor. La această materie nu faceți datele din interiorul claselor publice, doar funcțiile:

1. Este împotriva principiului deascundere a datelor

2. O să picați la POO!

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```
// Exemplu de clasă în C++ care nu se preteaza pentru acest curs!!
class Masina {
public: //la curs daca faceti astea publice luati 2
   std::string marca;
   int anFabricatie;
   void afisareDetalii() {
       std::cout << "Marca: " << marca << ", An fabricatie: " << anFabricatie
<< std::endl;
   }
};

// Crearea unui obiect de tip Masina
Masina masina1;
masina1.marca = "BMW";
masina1.anFabricatie = 2020;
masina1.afisareDetalii(); // Afiseaza "Marca: BMW, An fabricatie: 2020"
```

Incapsulare

Incapsularea este un concept central în POO care se referă la încorporarea datelor și a funcțiilor asociate cu acestea într-o singură unitate (obiect sau clasă). Accesul la datele și funcțiile incapsulate este controlat de către obiect, permițând protejarea datelor și ascunderea implementării.

```
// Exemplu de incapsulare
class Persoana {
private:
   std::string nume;
   int varsta;
public:
   void setNume(std::string n) {
       nume = n;
   }
   std::string getNume() {
       return nume;
   }
   void setVarsta(int v) {
       varsta = v;
   }
   int getVarsta() {
       return varsta;
   }
};
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Agregare (Compunere)

Agregarea este un concept în care o clasă conține un alt obiect ca membru, iar obiectul membru poate exista independent de clasa care îl conține. Știm că e vorba de această relație atunci când putem spune că O CLASĂ **ARE** ALTĂ CLASĂ

De exemplu: O mașină are un motor (Compunem clasele Motor și Mașină)

```
// Exemplu de agregare
class Motor {
public:
   std::string tip;
   int putere;
};

class Masina {
public:
   std::string marca;
   int anFabricatie;
   Motor motor; // Membru de tip Motor
};
```

În acest exemplu, clasa `Masina` conține un obiect de tip `Motor` ca membru.

Este bine să știți că, chiar dacă la nivel de cod arată mai mult sau mai puțin la fel, compoziția și agregarea se referă la lucruri puțin diferite.

Pentru agregare puteți considera exemplul: În clasa Facultate avem un string de Angajați. Angajații sunt în facultate, dar e mult spus că aparțin facultății.

Pentru compunere, relația trebuie să fie mai strânsă: În clasa Mașină avem un Motor. Motorul este o componenta esențială a mașinii și îl putem privi ca „aparținând” mașinii.

Pentru acest curs nu este necesar să faceți această distincție. Puteți folosi termenii interschimbabil.

Moștenire

Moștenirea este un concept prin care o clasă poate să își extindă funcționalitatea și să împărtășească datele și funcțiile de bază cu alte clase. Știm că e vorba de această relație atunci când putem spune că O CLASĂ **ESTE UN FEL DE** ALTĂ CLASĂ

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

De exemplu: O mașină are un motor (Un manager este un fel de angajat). Moștenim în clasa Manager toate proprietățile unui Angajat

```
// Exemplu de moștenire
class Angajat {
public:
   std::string nume;
   int salariu;
};

class Manager : public Angajat {
public:
   std::string departament;
};
```

În acest exemplu, clasa `Manager` moștenește membrii și funcțiile clasei de bază `Angajat`.

Polimorfism și Șabloane

Polimorfismul este un concept care permite obiectelor de același tip să fie tratate în mod similar, indiferent de clasa lor reală. Acest lucru se poate realiza prin intermediul funcțiilor virtuale și a suprascrierii acestora în clasele derivate.

Șabloanele (template-urile) sunt o caracteristică a C++ care permite crearea de funcții și clase generice care pot fi utilizate cu orice tip de date.

(nu vom da un exemplu aici, vom ajunge la asta spre finalul tutoriatelor)

Niveluri de accesibilitate: Public vs Private vs Protected

În limbajul de programare C++, există trei niveluri de accesibilitate pentru membrii unei clase: public, private și protected. Aceste niveluri de accesibilitate controlează modul în care membrii clasei pot fi accesați din alte părți ale codului.

Public

Membrii declarați ca publici sunt accesibili din orice parte a programului, inclusiv din afara clasei. Aceștia sunt utilizate pentru a expune funcționalități sau date ale clasei către alte părți ale programului. Un struct are membrii, by default, publici.

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```cpp
class Persoana {
public:
    std::string nume;
    int varsta;
    void afisareDetalii() {
        std::cout << "Nume: " << nume << ", Varsta: " << varsta << std::endl;
    }
};

int main() {
    Persoana p;
    p.nume = "Alex";
    p.varsta = 30;
    p.afisareDetalii(); // Acces la membrul public și la metoda publică
    return 0;
}
```

Private

Membrii declarați ca privați sunt accesibili numai din interiorul clasei respective. Aceștia nu pot fi accesați direct din alte părți ale programului și sunt utilizate pentru a ascunde implementările interne ale clasei. O clasă are membrii, by default, privați.

```cpp
class ContBancar {
private:
    std::string IBAN;
    float sold;
public:
    void depune(float suma) {
        sold += suma;
    }
};
```

Protected (vom discuta mai târziu despre asta)

Membrii declarați ca protejați sunt accesibili din interiorul clasei respective și din clasele derivate. Aceștia sunt utilizați în cadrul moștenirii pentru a permite accesul la membrii de bază ai clasei de la clasele derivate.

Operatorul de Scop și Rezoluție în C++

În limbajul de programare C++, operatorul de scop (`::`) este utilizat pentru a accesa membrii unei clase sau a unui spațiu de nume (namespace). Acesta permite specificarea explicită a contextului din care se dorește accesarea unui membru.

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Accesarea Membrilor Clasei

Operatorul de scop este folosit pentru a accesa membrii unei clase, inclusiv variabile, funcții și tipuri de date definite în interiorul clasei.

```
class Persoana {
public:
    static int contor; // membru static
    void afisareNume(const std::string& nume) {
        std::cout << "Nume: " << nume << std::endl;
    }
};

// Definirea membrului static al clasei Persoana
int Persoana::contor = 0;

int main() {
    Persoana p;
    p.afisareNume("John"); // Acces la membrul afisareNume() al clasei
Persoana
    std::cout << "Contor: " << Persoana::contor << std::endl; // Acces la
membrul static contor al clasei Persoana
    return 0;
}
```

În acest exemplu, `::` este utilizat pentru a accesa membrul static `contor` al clasei `Persoana`.

Rezoluția Calificatorului de Tip

Operatorul de rezoluție a membrilor (`::`) poate fi folosit și pentru a accesa membrii unei clase din clasele părinte (în moștenire multiplă), sau pentru a accesa membrii unui tip în cazul suprapunerii numelor. (Nu trebuie să rețineți neapărat, vom discuta în alt „episod”)

```
class A {
public:
    void afisare() {
        std::cout << "A" << std::endl;
    }
};

class B {
public:
    void afisare() {
        std::cout << "B" << std::endl;
    }
};
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```
class C : public A, public B {
public:
    void afisare() {
        A::afisare(); // Accesarea metodei afisare() din clasa A
        B::afisare(); // Accesarea metodei afisare() din clasa B
    }
};

int main() {
    C c;
    c.afisare();
    return 0;
}
```

În acest exemplu, `::` este utilizat pentru a accesa metoda `afisare()` din clasele de bază `A` și `B` în clasa derivată `C`.

```
// Exemplul din curs

#include <iostream>
using namespace std;

#define SIZE 100

// This creates the class stack.
//Exemplu din curs

class stack {
    int stek[SIZE];
    int tos;
public:
    void init();
    void push(int i);
    int pop();
};

void stack::init() {
    tos = 0;
}

void stack::push(int i) {
    if (tos == SIZE) {
        cout << "Stack is full.\n";
        return;
    }
    stek[tos] = i;
    tos++;
}

int stack::pop() {
    if (tos == 0) {
        cout << "Stack underflow.\n";
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```
        return 0;
    }
    tos--;
    return stek[tos];
}

int main() {
    stack stack1, stack2; // create two stack objects

    stack1.init();
    stack2.init();

    stack1.push(1);
    stack1.push(2);
    stack1.push(3);
    stack1.push(4);

    cout << stack1.pop() << endl;
    cout << stack1.pop() << endl;
    cout << stack1.pop() << endl;
    cout << stack1.pop() << endl;

    return 0;
}
```