Facultatea de Matematică și Informatică,
Universitatea din București

Tutoriat 2

Programare Orientată pe Obiecte - Informatică, Anul I

Tudor-Gabriel Miu
Radu Tudor Vrînceanu
3-22-2024

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>

Fundamentele Programării Orientate pe Obiecte în C++

Class

În C++, o clasă este un șablon pentru obiecte. Aceasta definește proprietățile și comportamentul obiectelor care vor fi create din acea clasă. Sintaxa pentru definirea unei clase în C++ este relativ simplă și implică utilizarea cuvântului cheie `class`, urmat de numele clasei și de blocul de membri ai clasei între acolade `{}`.

Sintaxa unei clase în C++

Sintaxa unei clase în limbajul de programare C++ este formată din câteva elemente cheie, care sunt urmate de o listă de membri și funcții ale clasei. Iată cum arată o declarație de bază a unei clase în C++:

1. class `NumeClasa { }` - Declarația începe cu cuvântul cheie class, urmat de numele clasei și un bloc de cod între acolade {} care conține membrii și funcțiile clasei.
2. `public:`, `protected:` și `private:` - Aceste cuvinte cheie definesc nivelurile de accesibilitate ale membrilor și funcțiilor clasei. Membrii și funcțiile declarați sub public: sunt accesibili din afara clasei, în timp ce cei declarați sub private: sunt accesibili doar din interiorul clasei.
3. Membrii clasei - Aceștia pot fi variabile (numite și câmpuri sau proprietăți) sau funcții (numite și metode). Aceștia sunt declarați cu un anumit tip și un nume.
4. Funcții membre - Acestea sunt funcțiile care operează asupra membrilor clasei. Acestea pot fi funcții publice, care sunt accesibile din exteriorul clasei, sau funcții private, care sunt accesibile numai din interiorul clasei.
5. Clase membre - Acestea sunt clase definite în interiorul altei clase. Acestea sunt utilizate pentru a organiza și a încapsula logică în cadrul clasei principale. Sintaxa pentru declararea unei clase membre în C++ este similară cu cea pentru o clasă obișnuită, dar trebuie să fie definită în interiorul clasei principale.

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

```cpp
#include <iostream>

class NumeClasa {
    // Membrii și funcții private
    TipMembru3 numeMembru3;
    TipMembru4 numeMembru4;

    // Funcții membre private
    TipReturn FunctiePrivata1(TipParametru3 parametru3) {
        // Corpul funcției
    }

    TipReturn FunctiePrivata2(TipParametru4 parametru4) {
        // Corpul funcției
    }

    // Clase membre
    class NumeClasaMembraPrivata {
        // membrii ai clasei membre
    };

public:
    // Membrii și funcții publice
    TipMembru1 numeMembru1;  //nu folositi date publice la acest curs
    TipMembru2 numeMembru2;  //nu folositi date publice la acest curs

    // Funcții membre publice
    TipReturn FunctiePublica1(TipParametru1 parametru1) {
        // Corpul funcției
    }

    TipReturn FunctiePublica2(TipParametru2 parametru2) {
        // Corpul funcției
    }

    // Clase membre
    class NumeClasaMembraPublica {
        // membrii ai clasei membre
    };
};
```

Declararea unui obiect

Declararea unui obiect în C++ este o operațiune simplă și presupune crearea unei instanțe a unei clase. Pentru a declara un obiect, folosim sintaxa:

```cpp
NumeClasa numeObiect;
```

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

De asemenea, se poate declara un obiect și imediat după declararea clasei, astfel:

```
class NumeClasa{

   // date si functii membre

}numeObiect;
```

Struct

În limbajul de programare C++, struct este un cuvânt cheie utilizat pentru a defini o structură de date. Exceptând sintaxa (`struct` VS `class`), unica difență între struct și class o reprezintă nivelul de accesibilitate default:

- `class` are membrii `private` by default
- `struct` are membrii `public` by default

```
struct NumeStructura{

   // date si functii membre, by default publice

private:

   // puteti declara date si functii membre private asa

}numeObiect;
```

Întrucât alte diferențe nu mai există, nu vom mai detalia lucrurile și pentru struct, dar trebuie să știți că nu este o practică bună să folosiți struct în loc de class (codul devine mai greu de citit când declari structuri cu toți membrii privați și clase cu toți membrii publici - spaghetti code de genul ăsta veți vedea la examen). Din acest motiv, la acest curs, nu este acceptat să folosiți struct pentru clase.

Union

În limbajul de programare C++, union este un tip de date care permite stocarea mai multor tipuri de date în aceeași locație de memorie. Cu alte cuvinte, o uniune permite suprapunerea datelor de tipuri diferite, dar ocupă spațiul necesar pentru cel mai mare dintre acestea.

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

By default, membrii unui union sunt publici, la fel ca la struct.

```
union NumeUnion{

   // date si functii membre, by default publice

};
```

Membrii unei uniuni sunt variabilele care sunt suprapuse în aceeași locație de memorie.

Toți membrii împart același spațiu de memorie, astfel încât schimbările făcute într-un membru pot afecta ceilalți membri.

Membrii uniunii pot fi accesați folosind operatorul punct `.` sau operatorul săgeată `->`, în funcție de modul în care este definită uniunea (ca o variabilă sau ca un pointer către uniune).

```
#include <iostream>

union Uniune{
   int x;
   float y;
   char z;

   void print() {
       std::cout << this->x << " " << this->y << " " << this->z <<
std::endl;
   }
};


int main() {

   Uniune u;
   u.z = 'a';
   u.y = 3.14;
   u.x = 5;

   u.print();

   return 0;
}
```

Ce se afișează?

```
5 7.00649e-45 ♣
```

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

Alte proprietăți:

1. Union nu poate moșteni și nu se poate moșteni din union
2. Nu putem avea funcții virtuale
    - Funcțiile virtuale sunt o caracteristică a claselor și sunt utilizate în polimorfismul dinamic, care nu este aplicabil în cazul uniunilor.
3. Nu avem variabile de instanță statice
    - Variabilele statice sunt acele variabile care sunt partajate între toate instanțele clasei și există într-un singur exemplar. O variabilă statică ar suprascrie tot unionul, făcându-l inutil.
4. Nu avem referințe în union
    - Referințele sunt adrese alternative pentru obiecte existente. O referință are un tip de date la care referă, iar în cazul union avem date de potențial tipuri diferite la aceeași adresă.
5. Nu avem obiecte care fac overload pe =
    - Acest lucru se datorează comportamentului ambiguu al suprapunerii datelor în uniuni și dificultății de a asigura o copiere corectă a datelor între obiecte.
6. Obiecte cu constructor/destructor definiți nu pot fi membri în union
    - Constructorii și destructorii implică inițializarea și distrugerea resurselor, care poate fi problematică în cadrul uniunilor, având în vedere că uniunea poate conține diferite tipuri de date.

Nu intrăm în foarte multe detalii. Unele dintre aceste concepte n-au fost încă explicate, dar nu vom mai reveni la union și e bine să existe aceste explicații undeva.

Union anonime

În limbajul de programare C++, o "uniune anonimă" este o uniune fără un nume explicit. Aceasta este adesea utilizată în cazul în care o uniune este utilizată ca membru al unei alte structuri sau clase și nu necesită un nume distinct.

Iată un exemplu de utilizare a unei uniuni anonime într-o structură:

```
struct ExempluStructura {
   int tip;
   union {
       int valoareInt;
       float valoareFloat;
   };
};
```

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

În acest exemplu, uniunea este declarată fără un nume explicit și este inclusă direct în structură. Astfel, membrii uniunii (`valoareInt` și `valoareFloat`) pot fi accesați direct ca membri ai structurii `ExempluStructura`.

Neavând un nume, nu putem declara obiecte de tipul respectiv. Într-un union anonim nu putem avea funcții și nici atribute private/protected.

Unionurile anonime declarate global **trebuie** să fie statice. Prin adăugarea cuvântului cheie `static`, se asigură că uniunea anonimă este vizibilă doar în cadrul fișierului sursă în care este definită.

```
static union {
   int a;
   char b;
};

int main() {

   return 0;
}
```

Funcții prieten (friend)

În limbajul de programare C++, o funcție prieten (sau friend function) este o funcție externă unei clase care are acces la membrii privați și protejați ai acelei clasei. Aceasta înseamnă că o funcție prieten poate accesa și modifica direct membrii privați și protejați ai unei clase, fără a fi necesar să utilizeze metodele publice ale clasei.

Pentru a declara o funcție externă ca prietenă a unei clase, se utilizează declarația friend în interiorul clasei respective. Iată cum arată sintaxa:

```
class Clasa {
private:
   int membruPrivat;

public:
   Clasa(int val) : membruPrivat(val) {}

   // Declarare de funcție prieten în interiorul clasei
   friend void afisareMembruPrivat(Clasa obiect);
};
```

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

În acest exemplu, `afisareMembruPrivat` este o funcție declarată ca prietenă a clasei `Clasa`. Acest lucru înseamnă că `afisareMembruPrivat` poate accesa și afișa direct membrul privat `membruPrivat` al obiectelor de tip `Clasa`.

Iată cum arată definiția funcției prieten în afara clasei:

```
void afisareMembruPrivat(Clasa obiect) {
    std::cout << "Membrul privat este: " << obiect.membruPrivat << '\n';
}
```

Observăm că această funcție este definită în afara clasei și are acces la membrii privați ai clasei `Clasa` datorită declarației `friend`.

Funcții prieten pentru mai multe clase

O funcție prietenă poate fi, de asemenea, declarată ca prietenă pentru mai multe clase. Iată cum puteți face acest lucru:

```
#include <iostream>

class B;

class A {
    int x;

public:
    A(int val) : x(val) {}

    friend void afisareAB(A objA, B objB);
};

class B {
    int y;

public:
    B(int val) : y(val) {}

    friend void afisareAB(A objA, B objB);
};

void afisareAB(A objA, B objB) {
    std::cout << "Valoarea lui x din A este: " << objA.x << std::endl;
    std::cout << "Valoarea lui y din B este: " << objB.y << std::endl;
}

int main() {
    A objA(5);
    B objB(10);

    afisareAB(objA, objB);
```

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

```
    return 0;
}
```

Observație: În codul dat, linia 3 declară o clasă numită "B". Aceasta este o declarație de clasă înaintea definiției efective a clasei B, care apare mai târziu în cod. Este o practică comună să declarați clase și funcții înainte de a le defini pentru a putea folosi acele declarații în alte părți ale codului. Acest lucru este necesar atunci când aveți nevoie să folosiți obiecte ale claselor respective în alte clase sau funcții care sunt definite înaintea acestora.

Funcții prieten din alte obiect

Așa cum am menționat, funcțiile friend sunt declarate în afara clasei pentru care sunt prietene. Din acest motiv, puteți găsi o funcție prietenă declarată într-un alt obiect. ca în exemplul următor:

```
#include <iostream>

class C2;

class C1 {
    int x;

    public:
        C1(int x) : x(x) {}

    void afisareC2(C2 c2);
};

class C2 {
    int y;

    public:
        C2(int y) : y(y) {}

    friend void C1::afisareC2(C2 c2);
};

void C1::afisareC2(C2 c2) {
    std::cout << c2.y << std::endl;
}

int main() {
    C1 c1(5);
    C2 c2(10);

    c1.afisareC2(c2);

    return 0;
}
```

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

Clase prietene

În C++, o clasă poate fi declarată ca fiind prietenă în altă clasă, ceea ce înseamnă că acea clasă prietenă are acces la membrii privați și protejați ai primei clase. Aceasta permite unei clase să acorde acces privilegiat altei clase pentru a manipula membrii săi, fără a le face publici sau a folosi metode de acces.

Iată cum puteți declara și utiliza o clasă prietenă în C++:

```cpp
class ClasaB; // Declarație înainte pentru a putea fi utilizată în ClasaA

class ClasaA {
private:
    int x;

public:
    ClasaA(int val) : x(val) {}

    // Declararea clasei prieten în interiorul clasei ClasaA
    friend class ClasaB;
};

class ClasaB {
public:
    void afisareX(ClasaA objA) {
        // Avem acces la membrul privat x al clasei ClasaA aici
        std::cout << "Valoarea lui x este: " << objA.x << std::endl;
    }
};
```

În acest exemplu, `ClasaB` este declarată ca fiind prietenă în interiorul clasei `ClasaA`. Aceasta înseamnă că `ClasaB` are acces la membrii privați ai clasei `ClasaA`. În funcția `afisareX` din `ClasaB`, putem accesa și afișa membrul privat `x` al obiectului de tip `ClasaA`

Funcții inline

Funcțiile inline sunt funcții definite în C++ care sunt expandate de către compilator în locul apelurilor lor, în loc să fie tratate ca funcții separate. Aceasta poate duce la o eficiență crescută în timpul execuției programului, deoarece nu se face un apel de funcție și nu se fac operații suplimentare pentru a salva și a restabili contextul funcției.

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

Pentru a defini o funcție inline, utilizați calificatorul inline înaintea definiției funcției. În general, funcțiile inline sunt utilizate pentru funcții scurte și simple, care sunt apelate frecvent, precum getteri și setteri, funcții de comparare etc.

Iată un exemplu simplu de o funcție inline:

```cpp
#include <iostream>

// Definirea funcției inline
inline int aduna(int a, int b) {
    return a + b;
}

int main() {
    int rezultat = aduna(3, 4); // Apelul funcției inline
    std::cout << "Rezultatul adunării este: " << rezultat << std::endl;
    return 0;
}
```

Acesta este un exemplu de funcție explicit inline (am folosit cuvântul cheie `inline`).

În C++, funcțiile definite în interiorul clasei sunt implicit considerate ca fiind inline. Aceasta înseamnă că funcțiile membru, definite direct în corpul unei clase, sunt tratate ca fiind inline de către compilator. Nu este necesar să folosiți explicit calificatorul `inline` în acest caz.

```cpp
class Clasa {
private:
    int x;

public:
    // Funcție membru definită implicit ca inline
    void setX(int val) {
        x = val;
    }

    void afisareX() {
        std::cout << "Valoarea lui x este: " << x << std::endl;
    }
};
```

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

Pentru a evita acest comportament, putem defini funcția în afara clasei.

```
class Clasa {
private:
   int x;

public:
   void setX(int val); // Declararea funcției în interiorul clasei
   void afisareX();
};

// Definirea funcției în afara clasei
void Clasa::setX(int val) {
   x = val;
}

void Clasa::afisareX() {
   std::cout << "Valoarea lui x este: " << x << std::endl;
}
```

Dacă vreți ca o funcție definită în afara clasei să fie inline, folosiți calificatorul `inline`, ca mai devreme.

Facem o scurtă paranteză (nu trebuie să rețineți dacă nu vreți, nu o să vă ajute la acest curs):

De ce ai vrea ca funcția să nu fie `inline`? Descrierea de mai devreme face să pară că e mai bine să fie, nu?

Există câteva motive pentru care ai putea dori să eviți ca o funcție membru să fie tratată ca fiind `inline` în C++:

1. Dimensiunea codului executabil: Funcțiile inline sunt expandate direct la locul apelului lor în codul executabil. Aceasta poate duce la creșterea dimensiunii codului executabil, în special dacă funcțiile inline sunt lungi sau sunt apelate frecvent în mai multe locuri din program.
2. Creșterea complexității codului: Funcțiile inline pot face codul să devină mai greu de înțeles, deoarece logica funcției este dispersată în întregul cod și nu este concentrată într-o singură locație (cum ar fi definiția funcției în afara clasei). Acest lucru poate face depanarea și întreținerea codului mai dificilă.
3. Optimizarea compilatorului: Compilatorul poate lua decizii mai bune de optimizare a codului dacă funcțiile nu sunt tratate ca fiind inline. Compilatorul poate decide să optimizeze funcțiile în mod diferit în funcție de circumstanțe, iar forțarea funcțiilor să fie inline poate restricționa aceste optimizări.

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

4. Cache-ul de instrucțiuni: Funcțiile inline pot duce la o mai mare fragmentare a cache-ului de instrucțiuni, deoarece codul funcției este duplicat în mai multe locuri. Acest lucru poate afecta performanța programului, în special pe arhitecturi cu cache-uri mici sau cu o arhitectură de procesor specifică.

În general, folosirea funcțiilor inline ar trebui să fie rezervată pentru situațiile în care performanța este critică și când funcțiile sunt foarte mici și simple. În alte cazuri, este de obicei mai bine să lăsați compilatorul să decidă dacă să trateze funcțiile ca fiind inline sau nu.

Constructori / Destructori

Constructorii și destructorii sunt funcții speciale în limbajul de programare C++ care sunt utilizate pentru inițializarea și eliberarea resurselor asociate obiectelor. Acestea sunt esențiale în programarea orientată pe obiecte pentru a asigura gestionarea corespunzătoare a resurselor și pentru a stabili un comportament consistent în timpul vieții obiectelor. Aceștia nu pot fi moșteniți, dar pot fi apelați din clasele derivate. Nu putem utiliza pointeri către aceștia.

Constructori:

Ce sunt constructorii: Constructorii sunt funcții membre ale unei clase care sunt invocate automat atunci când un obiect este creat. Acestea sunt folosite pentru inițializarea membrilor clasei și pentru a executa alte inițializări necesare înainte de utilizarea obiectului.

Tipuri de constructori: Există mai multe tipuri de constructori:

- **Constructor implicit:** Nu primește niciun parametru și este invocat atunci când un obiect este creat fără argumente.
- **Constructor cu parametri:** Primește unul sau mai mulți parametri și este utilizat pentru a inițializa obiectele cu anumite valori la crearea lor.
- **Constructor de copiere:** Este utilizat pentru a crea o copie a unui obiect existent.

Sintaxa constructorilor: Constructorii au același nume ca și clasa și nu au tip de returnare explicit. Sunt declarați și definiți ca orice altă funcție membru a clasei.

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

Destructori:

Ce este un destructor: Destructorul este o funcție membru a unei clase care este invocată automat atunci când un obiect este distrus. Este folosit pentru eliberarea resurselor pe care obiectul le-a alocat în timpul vieții sale.

Utilitatea destructorilor: Destructorii sunt folosiți pentru a evita scurgerile de memorie și pentru a elibera resurse precum memorie alocată dinamic, deschideri de fișiere, conexiuni de rețea etc.

Sintaxa destructorilor: Destructorul are același nume ca și clasa, precedat de caracterul „~” (tilde). Nu are niciun parametru și nu returnează nimic.

Destructorul este apelat automat de către sistemul de operare C++ atunci când obiectul își pierde contextul (se încheie funcția în care a fost declarat local sau se încheie programul pentru cele globale) sau atunci când este dezalocat spațiul de memorie asociat cu obiectul. Acest lucru se întâmplă atunci când un obiect este eliminat din scopul său, fie automat (la sfârșitul unei funcții), fie manual (prin ștergerea explicită a unui pointer către acel obiect).

Exemplu de clasă cu constructori și destructori definiți de către noi:

```cpp
#include <iostream>

class Exemplu {
    int x;

public:
    Exemplu() {     // Constructor implicit
        x = 0;
        std::cout << "Constructor implicit apelat" << std::endl;
    }

    Exemplu(int val) {     // Constructor cu parametru
        x = val;
        std::cout << "Constructor cu parametru apelat" << std::endl;
    }

    ~Exemplu() {     // Destructor
        std::cout << "Destructor apelat" << std::endl;
    }
};

int main() {
    // Crearea și distrugerea obiectelor
    Exemplu obiect1; // Constructor implicit apelat
    Exemplu obiect2(5); // Constructor cu parametru apelat
    return 0;
} // Destructor apelat pentru fiecare obiect la sfârșitul funcției main, in
care au fost create obiectele
```

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

Este importantă rescrierea acestora atunci când alocăm dinamic memoria. Dacă în constructor se aloca dinamic memorie (utilizăm `new`), în destructor va fi necesar să o dealocăm cu `delete`. Vezi exemplul de mai jos:

```
class Exemplu {
   int *pointer;
public:
   Exemplu() {     // Constructorul
       pointer = new int(0); // Inițializăm pointerul cu o valoare
implicită
   }
   ~Exemplu() {    // Destructorul
       delete pointer; // Dezalocăm memoria alocată pentru pointer
   }
};
```

Constructorul de copiere

Constructorul de copiere în C++ este un constructor special care este utilizat pentru a crea o copie a unui obiect existent. Acesta este invocat atunci când un nou obiect este inițializat cu un alt obiect de același tip. Constructorul de copiere creează o copie a obiectului specificat ca argument și inițializează membrii noului obiect cu valorile membrilor obiectului furnizat ca parametru.

```
#include <iostream>


class Exemplu {
private:
   int x;


public:
   // Constructor implicit
   Exemplu() {
       x = 0;
   }


   // Constructor de copiere
   Exemplu(const Exemplu &altObiect) {
       x = altObiect.x;
       std::cout << "Constructor de copiere apelat" << std::endl;
   }


   // Metoda pentru afișarea valorii membrului x
   void afisareX() {
```

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

```
        std::cout << "Valoarea lui x este: " << x << std::endl;
    }
};


int main() {
    Exemplu obiect1;
    obiect1.afisareX(); // Afiseaza: Valoarea lui x este: 0


    // Crearea obiectului 2 prin copierea obiectului 1
    Exemplu obiect2 = obiect1;
    obiect2.afisareX(); // Afiseaza: Valoarea lui x este: 0


    return 0;
}
```

Constructorul de copiere e folosit la inițializări. Dacă vrem, mai târziu, să scriem `obiect1 = obiect 2`, se va face o atrbuire. Există un operator de atribuire de care o să vorbim imediat.

Orice clasă are, by default:

- un constructor de inițializare
- un constructor de copiere
- un destructor
- un operator de atribuire

Programatorul poate decide să redefinească unele dintre aceste funcții sau pe toate. Trebuie să fim atenți: de cele mai multe ori redefinirea uneia schimbă suficiente lucruri cât să fie necesar să le redefinim și pe celelalte.

By default, constructorul de copiere va copia, **bit cu bit**, datele membre din obiectul copiat. Când ar putea să ne pună probleme acest aspect?

Când vine vorba de pointeri și adrese de memorie!

```
#include <iostream>

class Exemplu {
    int *pointer;

public:
    Exemplu() {     // Constructorul
        pointer = new int(0); // Inițializăm pointerul cu o valoare
implicită
    }

    ~Exemplu() {     // Destructorul
        delete pointer; // Dezalocăm memoria alocată pentru pointer
```

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

```
    }

    void afiseazaValoare() {     // Metodă pentru afișarea valorii
        std::cout << "Valoare: " << *pointer << std::endl;
    }

    void setValoare(int valoare) {     // Metodă pentru setarea unei valori
        *pointer = valoare;
    }
};

int main() {
    Exemplu obiect1; // Creăm un obiect
    obiect1.afiseazaValoare(); // Afișăm valoarea inițială
    Exemplu obiect2 = obiect1; // Copiem obiectul
    obiect2.afiseazaValoare(); // Afișăm valoarea obiectului copiat

    obiect1.setValoare(42); // Modificăm valoarea obiectului original
    obiect1.afiseazaValoare(); // Afișăm valoarea obiectului original
    obiect2.afiseazaValoare(); // Problema: S-a modificat și valoarea
obiectului copiat

    return 0;
}
```

Mai rău, haideți să vedem ce se întâmplă dacă transmitem un obiect de tip `Exemplu` ca parametru al unei funcții:

```
#include <iostream>

class Exemplu {
private:
    int *pointer;

public:
    // Constructorul
    Exemplu() {
        pointer = new int(0); // Inițializăm pointerul cu o valoare
implicită
    }

    // Destructorul
    ~Exemplu() {
        delete pointer; // Dezalocăm memoria alocată pentru pointer
    }

    // Metodă pentru afișarea valorii
    static void afiseazaValoare(Exemplu obiect) {
        std::cout << "Valoare: " << *obiect.pointer << std::endl;
    }

};

int main() {
    Exemplu obiect1; // Creăm un obiect
```

```
    Exemplu::afiseazaValoare(obiect1); // Afisăm valoarea initială (0)
    Exemplu::afiseazaValoare(obiect1); // Afisăm valoarea obiectului
(82517072)

    return 0;
}
```

1. Așa cum am mai discutat, la apelarea unei funcții cu parametrii transmiși prin valoare, compilatorul face copii locale parametrilor. În acest caz, se folosește, deci, constructorul de copiere.
2. Local, la nivelul funcției `afișeazăValoare` există acum o copie a obiectului `obiect1`. Țineți minte că pointerul a fost copiat bit cu bit deoarece nu am suprascris constructorul de copiere, deci arată arată către adresa din `obiect1`.
3. La ieșirea din funcția se apelează destructorii obiectelor locale. Se distruge copia făcută și, implicit, pointerul din obiectul copiat, dar acesta coincide cu cel din `obiect1`. Tocmai ce am distrus pointerul din obiectul inițial.
4. La al doilea apel pointerul nu mai arată spre ce trebuie și obținem o valoare arbitrară. Mai mult, programul se încheie cu `exit code -1073740940 (0xC0000374)` - STATUS_HEAP_CORRUPTION. De ce? Pentru că utilizăm o valoare dintr-o zonă dealocată. Acestă eraore poartă numele de „use after free”. La finalul main, programul apelează destructorul pentru `obiect1`. Acestă eroare este cunoscută ca „double free error”. Încercă să dealocăm o zonă deja dealocată.

Acestea fiind zise, e clar că, în acest caz, e nevoie să rescriem constructorul de copiere:

```
#include <iostream>
#include <cstring>

class Exemplu {
private:
    char *nume;

public:
    // Constructorul
    Exemplu(const char *nume) {
        this->nume = new char[strlen(nume) + 1];
        strcpy(this->nume, nume);
    }

    // Destructorul
    ~Exemplu() {
        delete[] nume;
    }

    // Constructorul de copiere personalizat
    Exemplu(const Exemplu &other) {
```

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

```
        this->nume = new char[strlen(other.nume) + 1];
        strcpy(this->nume, other.nume);
    }

    // Metoda pentru afișarea numelui
    void afiseazaNume() {
        std::cout << "Nume: " << nume << std::endl;
    }
};

int main() {
    Exemplu obiect1("John"); // Initializam un obiect
    Exemplu obiect2 = obiect1; // Copiem obiectul

    obiect1.afiseazaNume(); // Afisam numele obiectului initial
    obiect2.afiseazaNume(); // Afisam numele obiectului copiat

    return 0;
}
```

Acum merge bine.

Conversii explicite de date prin constructori

Avem un constructor cu un parametru ca în exemplul de mai jos:

```
#include <iostream>

class Celsius {
private:
    float temperatura;

public:
    // Constructor cu un parametru pentru a inițializa temperatura
    Celsius(float temp){
        temperatura = temp;
    }

    // Funcție pentru a afișa valoarea temperaturii în grade Celsius
    void afisare() {
        std::cout << "Temperatura este: " << temperatura << " grade Celsius"
<< std::endl;
    }
};
```

Putem crea un obiect și îi putem atribui o valoare direct:

```
int main() {
    // Crearea unui obiect Celsius cu o valoare de tip float
    Celsius celsiusObj = 25.5;
    // Apelarea funcției afisare pentru a afișa valoarea temperaturii
```

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

```
    celsiusObj.afisare();

    return 0;
}
```

Compilatorul va face conversia în acest caz și va crea un obiect de tip `Celsius` în care `temperatura = 25.5`. Aceasta este o conversie implicită. Pentru a evita acest comportament care poate părea ciudat, marcați constructorul cu `explicit`:

```
explicit Celsius(float temp){...}
```

Cuvântul cheie `explicit` împiedică compilatorul să folosească constructorul definit de noi pentru conversii implicite.

Lista de inițializare

Lista de inițializare în C++ este o metodă de inițializare a membrilor unei clase înainte de a intra în corpul constructorului. Aceasta este o parte a sintaxei constructorului și este folosită pentru a inițializa membrii clasei și pentru a specifica argumentele constructorului părintelui în moștenirea clasică. Utilizarea listei de inițializare este recomandată, deoarece permite inițializarea membrilor în mod eficient și ajută la evitarea unor comportamente nedorite.

Iată cum arată sintaxa unei liste de inițializare într-un constructor:

```
NumeClasa::NumeClasa(TipMembru1 valoare1, TipMembru2 valoare2, ...) :
membru1(valoare1), membru2(valoare2), ... {
    // Corpul constructorului
}
```

În exemplul de mai sus:

- `NumeClasa` este numele clasei pentru care este definit constructorul.
- `TipMembru1`, `TipMembru2`, etc., sunt tipurile membrilor clasei.
- `valoare1`, `valoare2`, etc., sunt valorile cu care sunt inițializați membrii clasei.
- `membru1`, `membru2`, etc., sunt membrii clasei care sunt inițializați în lista de inițializare.

Iată un exemplu concret pentru o clasă `Exemplu` cu un membru `x` care este inițializat în lista de inițializare a constructorului:

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

```cpp
#include <iostream>

class Exemplu {
private:
    int x;

public:
    // Constructor cu un singur parametru, inițializează membrul x folosind lista de inițializare
    Exemplu(int val) : x(val) {}

    void afisare() {
        std::cout << "Valoarea lui x este: " << x << std::endl;
    }
};

int main() {
    Exemplu obiect(42); // Inițializarea obiectului și a membrului său x folosind lista de inițializare
    obiect.afisare(); // Afișează valoarea membrului x
    return 0;
}
```

În acest exemplu, membrul `x` al clasei `Exemplu` este inițializat cu valoarea `val` în lista de inițializare a constructorului. Aceasta este o metodă eficientă și clară de a inițializa membrii clasei înainte de a intra în corpul constructorului.

Polimorfism și overload pe constructori

De cele mai multe ori aceste concepte sunt foarte utile când vine vorba de constructori. Am văzut deja că nu mereu ne dorim comportamentul predefinit, așa că vom face des overload pe constructori.

De asemenea, putem să ne dorim să inițializăm obiecte cu un singur parametru, doi, trei sau mai mulți. Vom folosi polimorfismul și vom avea constructori cu număr diferit de parametrii în cadrul aceleiași clase. Observați diferitele logici în constructorii de mai jos:

```cpp
class Exemplu {
private:
    int x;
    int y;
```

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

```
    int z;

public:
    // Constructor implicit
    Exemplu() {
        x = 0;
        y = 0;
        z = 0;
    }

    // Constructor cu un parametru
    Exemplu(int val) {
        x = val;
        y = val;
        z = val;
    }

    // Constructor cu doi parametri
    Exemplu(int val1, int val2) {
        x = val1;
        y = val2;
        z = 0;
    }

    // Constructor cu trei parametri
    Exemplu(int val1, int val2, int val3) {
        x = val1;
        y = val2;
        z = val3;
    }

};

int main() {
    Exemplu obiect1; // Apel constructor implicit
    Exemplu obiect2(10); // Apel constructor cu un parametru
    Exemplu obiect3(20, 30); // Apel constructor cu doi parametri
    Exemplu obiect4(40, 50, 60); // Apel constructor cu trei parametri

    return 0;
}
```

Un alt exemplu: Să definim un array de obiecte. Probabil nu le vom inițializa pe toate cu valorile pe care ni le dorim de la început, așa că e nevoie de un constructor fără parametrii care să creeze obiectele. Vom modifica datele din ele mai târziu. Obiectele create inițial sunt, practic, neinițializate.

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

```cpp
#include <iostream>

class Obiect {
private:
    int x;
    int y;

public:
    // Constructor implicit (fără parametrii)
    Obiect() {
        x = 0; // Inițializăm membrii cu valorile implicite
        y = 0;
    }

    // Metoda pentru a seta valorile membrilor
    void set(int valX, int valY) {
        x = valX;
        y = valY;
    }

    // Metoda pentru a afișa valorile membrilor
    void afisare() const {
        std::cout << "x: " << x << ", y: " << y << std::endl;
    }
};

int main() {
    const int lungimeTablou = 5;
    Obiect tablou[lungimeTablou]; // Inițializăm tabloul fără a furniza
valori inițiale

    // Modificăm valorile obiectelor din tablou
    for (int i = 0; i < lungimeTablou; ++i) {
        tablou[i].set(i + 1, i + 2);
    }

    // Afișăm valorile obiectelor din tablou
    for (int i = 0; i < lungimeTablou; ++i) {
        std::cout << "Elementul " << i << " din tablou: ";
        tablou[i].afisare();
    }

    return 0;
}
```

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

Ordinea de apel a constructorilor și destructorilor

Un exercițiu rapid:

```
#include <iostream>

class Exemplu {
public:
   Exemplu() {
       std::cout << "Constructor apelat" << std::endl;
   }

    Exemplu(const Exemplu &altObiect) {
       std::cout << "Constructor de copiere apelat" << std::endl;
   }

   ~Exemplu() {
       std::cout << "Destructor apelat" << std::endl;
   }

   void functieCareNuFaceNimic(const Exemplu obiect) {}

   static Exemplu functiCareRetureazaAcelasiObiect(const Exemplu obiect) {
       return obiect;
   }
};

int main() {
   Exemplu obiect1; //exemplu 1
   std::cout << std::endl;

   Exemplu obiect2 = obiect1; //exemplu2
   std::cout << std::endl;

   obiect1.functieCareNuFaceNimic(obiect2); //exemplu 3
   std::cout << std::endl;

   Exemplu::functiCareRetureazaAcelasiObiect(obiect2); //exemplu 4
   std::cout << std::endl;

   obiect2 = Exemplu::functiCareRetureazaAcelasiObiect(obiect1); //exemplu
5
   std::cout << "\nSfarsitul programului" << std::endl;

   return 0;
}
```

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

Ce afișează?

```
Constructor apelat

Constructor de copiere apelat

Constructor de copiere apelat
Destructor apelat

Constructor de copiere apelat
Constructor de copiere apelat
Destructor apelat
Destructor apelat

Constructor de copiere apelat
Constructor de copiere apelat
Destructor apelat
Destructor apelat

Sfarsitul programului
Destructor apelat
Destructor apelat
```

De ce nu e nicio diferență între exemplul 4 și exemplul 5? Chiar dacă în exemplul 4 nu salvăm ce s-a returnat, o copie se face oricum și se transmite către main. Se copiază și se distruge imediat.

Operatorul de atribuire (operator=)

Operatorul de atribuire (operator=) este un operator special în C++ care permite atribuirea valorilor dintr-un obiect în altul al aceleiași clase. Este folosit pentru a copia valorile membrilor unui obiect în altul sau pentru a face o atribuire a unei referințe către un obiect existent.

Dacă nu definim operatorul de atribuire pentru o clasă, compilatorul va genera unul implicit care va face o simplă copiere bit cu bit a tuturor membrilor.

Suprascriem operatorul= în aceleași tipuri de situații în care suprascriem și constructorul de copiere (când există probleme de management al memoriei). Sintaxa pentru suprascriere este evidențiată în următorul exemplu:

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

```cpp
#include <iostream>

class Exemplu {
private:
    int x;
    int y;

public:
    // Constructor implicit
    Exemplu() : x(0), y(0) {}

    // Constructor cu doi parametri
    Exemplu(int valX, int valY) : x(valX), y(valY) {}

    // Suprascrierea operatorului de atribuire
    Exemplu& operator=(const Exemplu& altObiect) {
        // Verificăm dacă nu încercăm să ne atribuim singuri

        x = altObiect.x;
        y = altObiect.y;

        return *this;
    }

    // Metoda pentru a afișa valorile membrilor
    void afisare() const {
        std::cout << "x: " << x << ", y: " << y << std::endl;
    }
};

int main() {
    // Creăm două obiecte și le atribuim valorile unul altuia
    Exemplu obiect1(10, 20);
    Exemplu obiect2;

    std::cout << "obiect1 inainte de atribuire: ";
    obiect1.afisare();
    std::cout << "obiect2 inainte de atribuire: ";
    obiect2.afisare();

    obiect2 = obiect1; // Atribuim valorile din obiect1 în obiect2

    std::cout << "\nobiect1 dupa atribuire: ";
    obiect1.afisare();
    std::cout << "obiect2 dupa atribuire: ";
    obiect2.afisare();

    return 0;
}
```

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

Pointerul `this`

În limbajul de programare C++, `this` este un pointer special către obiectul curent în cadrul căruia este apelat membrul unei clase. Acesta poate fi folosit pentru a accesa și modifica membrii obiectului curent. `this` este implicit disponibil în cadrul metodelor membru ale claselor și este utilizat pentru a face referire la instanța obiectului asupra căruia se operează.

Iată câteva utilizări comune ale atributului `this`:

1. **Accesarea membrilor obiectului:** `this` poate fi folosit pentru a accesa membrii obiectului curent în cadrul metodelor membru. Aceasta este utilă atunci când numele membrilor sunt ambigue în interiorul metodelor din cauza unor parametri cu același nume sau pentru a face referire clară la membrul obiectului.

```cpp
#include <iostream>

class Exemplu {
private:
    int x;

public:
    void setX(int x) {
        // Utilizarea this pentru a accesa membrul x al obiectului curent
        this->x = x;
    }

    int getX() const {
        // Utilizarea this pentru a accesa membrul x al obiectului curent
        return this->x;
    }
};

int main() {
    Exemplu obiect;
    obiect.setX(42);
    std::cout << "Valoarea lui x este: " << obiect.getX() << std::endl;
    return 0;
}
```

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

2. **Returnarea referinței către obiectul curent:** În unele cazuri, este util să returnăm referința către obiectul curent dintr-o metodă. Aceasta poate fi folosită pentru a permite cascading-ul apelurilor de metode pe același obiect.

```cpp
#include <iostream>

class Exemplu {
private:
    int x;

public:
    Exemplu& setX(int x) {
        this->x = x;
        return *this; // Returnează referința către obiectul curent
    }

    int getX() const {
        return this->x;
    }
};

int main() {
    Exemplu obiect;
    obiect.setX(42).setX(24); // Cascading-ul apelurilor de metode
    std::cout << "Valoarea lui x este: " << obiect.getX() << std::endl;
    return 0;
}
```

Utilizarea `static`

În limbajul C++, cuvântul cheie static poate fi folosit pentru a defini membrii clasei care aparțin întregii clase, în loc de instanțelor individuale ale acelei clase. Acest lucru înseamnă că o singură instanță a acelui membru este partajată între toate instanțele de obiecte ale clasei.

Iată cum poate fi utilizat `static` în diferite contexte în C++:

1. Variabile statice de clasă: Variabilele statice de clasă sunt declarate folosind cuvântul cheie `static` și aparțin întregii clase, nu unui obiect specific. Ele sunt inițializate o singură dată, indiferent de câte obiecte sunt create din acea clasă.

```cpp
#include <iostream>
```

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

```
class Exemplu {
   static int contor; // Variabilă statică de clasă

public:
   Exemplu() { //In acest caz am ales ca la fiecare apel al constructorului
sa se incrementeze variabila contor. Practic contorul va retine numarul de
obiecte create
       contor++;
   }

   // Metoda care returnează valoarea variabilei statice
   static int getContor() {
       return contor;
   }
};

// Inițializare a variabilei statice de clasă
int Exemplu::contor = 0;

int main() {
   // Crearea a două obiecte
   Exemplu e1, e2;

   std::cout << "Valoarea variabilei statice contor: " <<
Exemplu::getContor() << std::endl;

   return 0;
}
```

2. **Metode statice:** Metodele statice ale unei clase pot fi apelate direct prin intermediul clasei, fără a crea o instanță a acelei clase. Ele nu pot accesa membrii non-statici ai clasei, deoarece metodele statice nu sunt legate de o instanță specifică. Metoda `getContor()` de mai sus e statică. Un alt exemplu exem mai jos:

```
#include <iostream>

class Utilitare {
public:
   static int adunare(int a, int b) {
       return a + b;
   }
};

int main() {
   // Apelarea metodei statice adunare
   std::cout << "Rezultatul adunarii: " << Utilitare::adunare(5, 3) <<
```

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

```
std::endl;

    return 0;
}
```

Metodele statice nu au pointer `this`. Ele nu se referă la o instanță specifică a clasei și se pot referi doar la membrii statici din cadrul clasei.

Clase locale

În C++, clasele locale sunt clase definite în interiorul unor alte clase sau în interiorul unor funcții. Aceste clase sunt disponibile doar în cadrul contextului în care au fost definite și nu pot fi accesate din afara acestuia. Ele sunt utile pentru a organiza codul și pentru a ascunde implementările auxiliare în cadrul unor clase sau funcții mai mari.

Iată un exemplu simplu de utilizare a unei clase locale într-o funcție dintr-o altă clasă:

```
#include <iostream>

class ClasaExterioara {
public:
    void afisare() {
        // Clasa locală definită în interiorul metodei afisare()
        class ClasaLocala {
        public:
            void mesaj() {
                std::cout << "Salut din clasa locala!\n";
            }
        };

        ClasaLocala clasaLocala;
        clasaLocala.mesaj();
    }
};

int main() {
    ClasaExterioara obiect;
    obiect.afisare();

    return 0;
}
```

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

Pentru a putea accesa o funcție statică din clasa interioară e nevoie să o specifcați și pe cea exterioară. Puteți să vă gândiți la o analogie cu sistemul de fișiere din calculatorul vostru: ca să accesați un fișier aveți nevoie de întreaga cale până la el: disc, folder, folder, (...), folder, fișier.

```
#include <iostream>

class ClasaExterioara {
private:
   int x; // Membru al clasei exterioare

public:
   ClasaExterioara(int val) : x(val) {}

   // Clasa internă (sau clasă în clasă)
   class ClasaInterioara {
   public:
       void afisareX(const ClasaExterioara& obiect) {
           // Accesăm membrul privat x al clasei exterioare folosind
operatorul de rezoluție ::
           std::cout << "Valoarea lui x din clasa exterioara este: " <<
obiect.x << std::endl;
       }
   };
};

int main() {
   ClasaExterioara obiectExterior(10);
   ClasaExterioara::ClasaInterioara obiectInterior;

   // Apelăm metoda din clasa interioară pentru a accesa membrul privat x
al clasei exterioare
   obiectInterior.afisareX(obiectExterior);

   return 0;
}
```