Tiberiu-Teodor COCIAȘ Delia Elisabeta UNGUREANU

PROGRAMAREA ORIENTATĂ PE OBIECTE: O PARALELĂ PRACTICĂ ÎNTRE C++ ȘI PYTHON

Editura Universității Transilvania din Brașov
2025

EDITURA UNIVERSITĂȚII TRANSILVANIA DIN BRAȘOV

Adresa:	Str. Iuliu Maniu nr. 41A
	500091 Brașov
Tel.:	0268 476 050
Fax:	0268 476 051
E-mail:	editura@unitbv.ro

Editură recunoscută CNCSIS, cod 81

ISBN 978-606-19-1805-8 (ebook)

Lucrarea a fost avizată de Consiliul Departamentului de Automatică și tehnologia informației, Facultatea de Inginerie electrică și știința calculatoarelor a Universității Transilvania din Brașov.

Cuprins

1. Introducere în Programarea Orientată pe Obiecte

Prezentare generală a paradigmelor de programare
Beneficiile programării orientate pe obiecte (POO)
Rolul claselor și obiectelor în programarea orientată pe obiecte

1.1 Prezentare generală a paradigmelor de programare

Paradigmele de programare în dezvoltarea software se referă la diferitele abordări și filozofii care ghidează structura, design-ul și implementarea sistemelor software. Aceste paradigme oferă dezvoltatorilor principii și orientări pentru organizarea codului, gestionarea complexității și rezolvarea problemelor în mod eficient. Mai jos sunt câteva paradigme de programare recunoscute [14] [6] [7] [12]:

1. **Programare Imperativă**: Această paradigmă pune accentul pe descrierea unei secvențe de pași pe care programul trebuie să îi urmeze pentru a ajunge la o stare dorită. Se concentrează pe schimbarea stării programului prin instrucțiuni care modifică variabilele direct. Limbajele precum C și Pascal urmează această paradigmă îndeaproape.

2. **Programare Declarativă**: În contrast cu programarea imperativă, programarea declarativă se concentrează pe descrierea rezultatului dorit fără a specifica pașii exacți pentru a-l obține. Exemple includ Programarea Funcțională și Programarea Logică.

3. **Programare Funcțională**: Această paradigmă de programare care se bazează pe concepte din matematică, în special din teoria funcțiilor. Spre deosebire de programarea imperativă, unde accentul este pus pe modificarea stării și pe execuția de comenzi secvențiale, programarea funcțională promovează scrierea codului sub forma unor funcții pure, care pentru aceleași argumente vor returna mereu aceleași rezultate, fără efecte secundare. Programarea funcțională încurajează imutabilitatea, adică datele nu sunt modificate după ce au fost create, ci se creează versiuni noi în urma aplicării unor funcții. De asemenea, utilizează frecvent funcții de ordin superior (funcții

care acceptă alte funcții ca parametri sau returnează funcții) și recursivitatea, în locul structurilor repetitive tradiționale (precum buclele `for` sau `while`).

4. **Programare Orientată pe Obiecte (POO)**: POO organizează software-ul în jurul obiectelor și datelor în loc de acțiuni și logică. Accentuează încapsularea, moștenirea și polimorfismul pentru a structura codul. Limbaje precum Java, C++ și Python sunt frecvent utilizate pentru POO.

5. **Programare Procedurală**: Similară cu programarea imperativă, programarea procedurală se concentrează pe divizarea unui program în proceduri sau rutine mai mici. Accentuează apelurile de proceduri și design-ul modular. C este un exemplu elocvent de limbaj care susține programarea procedurală.

6. **Programare Orientată pe Evenimente**: Această paradigmă se concentrează pe răspunsul la evenimente generate de utilizatori sau sistem. De obicei, implică definirea de gestionari de evenimente sau funcții de apel înapoi pentru a executa acțiuni specifice în răspuns la evenimente. JavaScript, utilizat în dezvoltarea web, adesea urmează această paradigmă.

7. **Programare Orientată pe Aspecte (POA)**: POA își propune să crească modularitatea prin permiterea separării preocupărilor transversale, cum ar fi jurnalizarea, securitatea și gestionarea tranzacțiilor, de la logica principală a afacerii. Limbaje sau cadre POA cum ar fi AspectJ permit dezvoltatorilor să definească aspecte care traversează mai multe părți ale codului sursă.

8. **Programare Logică**: Programarea logică implică descrierea unui set de fapte și reguli și apoi formularea de interogări către un sistem care este capabil să deducă răspunsuri pe baza inferenței logice. Prolog este un limbaj bine-cunoscut care urmează această paradigmă.

9. **Programare Orientată pe Concurență**: Odată cu creșterea procesoarelor multinucleu și a sistemelor distribuite, programarea orientată pe concurență se concentrează pe scrierea codului care poate gestiona eficient mai multe sarcini care se execută simultan. Limbaje precum Erlang și Go oferă suport încorporat pentru concurență.

10. **Programare Orientată pe Date**: Această paradigmă se concentrează pe descrierea fluxului de date printr-un sistem, adesea folosind grafice de flux de date sau structuri similare. Accentuează transformările asupra datelor în locul fluxului de control explicit.

Aceste paradigme nu sunt excluzive între ele, iar multe limbaje de programare și cadre incorporează elemente din mai multe paradigme. Alegerea paradigmei de programare depinde adesea de factori precum natura domeniului problemei, cerințele de performanță, preferințele echipei și punctele forte și slabe ale diferitelor paradigme pentru o anumită sarcină.

1.2 Beneficiile programării orientate pe obiecte (POO)

Programarea Orientată pe Obiecte (POO) este esențială în dezvoltarea software-ului modern datorită capacității sale de a organiza codul eficient și de a modela entitățile din lumea reală în mod eficace. La baza sa, POO oferă o modalitate de structurare a programelor în jurul obiectelor, care sunt instanțe ale claselor ce conțin atât date (atribute), cât și comportament (metode). Această paradigmă este crucială pentru că oferă mai multe beneficii care ajută semnificativ în sarcinile de programare [2].

În primul rând, POO promovează modularitatea prin descompunerea sistemelor complexe în părți mai mici și mai ușor de gestionat, reprezentate de clase. Fiecare clasă încapsulează funcționalități specifice, permițând dezvoltatorilor să se concentreze pe componente individuale fără a fi copleșiți de întregul cod sursă. Această abordare modulară facilitează reutilizarea codului, deoarece clasele și obiectele pot fi ușor utilizate în diferite părți ale programului sau chiar în alte programe. Prin reutilizarea componentelor existente, dezvoltatorii pot economisi timp și efort, asigurând în același timp consistență și fiabilitate în proiecte.

În plus, POO încurajează scalabilitatea prin furnizarea unei structuri flexibile care poate adapta modificări și extinderi pe măsură ce programul evoluează. Noile funcționalități pot fi adăugate prin extinderea claselor existente sau crearea altora noi, fără a fi necesare modificări extinse ale codului sursă existent. Această scalabilitate face ca POO să fie potrivită în special pentru proiecte de mari dimensiuni, în care menținerea coerenței și adaptabilității codului este crucială [9].

Un alt avantaj semnificativ al POO este abstractizarea, care permite dezvoltatorilor să se concentreze pe ceea ce face un obiect în loc de modul în care îl face. Prin ascunderea detaliilor de implementare interne ale obiectelor în spatele interfețelor bine definite, POO permite dezvoltatorilor să lucreze la niveluri superioare de abstractizare, îmbunătățind citirea și menținerea codului. Această abstractizare simplifică procesul de dezvoltare și promovează colaborarea între membrii echipei, deoarece fiecare clasă oferă o interfață clară și standardizată pentru interacțiunea cu obiectele sale.

În plus, POO facilitează polimorfismul, capacitatea diferitelor obiecte de a răspunde la același mesaj (apel de metodă) în moduri diferite. Această caracteristică îmbunătățește flexibilitatea și extensibilitatea codului, permițând dezvoltatorilor să scrie cod generic care poate opera pe obiecte de diferite tipuri. Polimorfismul promovează reutilizarea codului și simplifică menținerea prin minimizarea necesității de instrucțiuni condiționale și verificări de tipuri explicite.

În concluzie, Programarea Orientată pe Obiecte este indispensabilă în programare datorită capacității sale de a organiza codul eficient, de a promova modularitatea, scalabilitatea, abstractizarea și polimorfismul. Adoptând principiile POO, dezvoltatorii pot crea sisteme software robuste, ușor de menținut și scalabile, care să răspundă cerințelor dezvoltării software moderne.

1.3 Rolul claselor și obiectelor în programarea orientată pe obiecte

Programarea Orientată pe Obiecte este o paradigma care se bazează pe conceptul de "obiecte" și interacțiunile acestora pentru a proiecta și construi sisteme software. În centrul POO stau clasele și obiectele, care servesc drept blocuri de construcție pentru structurarea și organizarea codului.

O clasă poate fi gândită ca un șablon sau un model (eng. template) pentru crearea obiectelor. Ea definește atributele (cunoscute și sub numele de variabile membre) și metode (cunoscute și sub numele de funcții membre) pe care toate obiectele acelei clase le vor avea. Un obiect este o instanță a unei clase. El reprezintă o realizare concretă a atributelor și metodelor definite de clasa sa. În termeni mai simpli, puteți gândiți o clasă ca pe un șablon, iar un obiect ca pe o copie specifică creată din acel șablon [1].

Mai jos se regăsesc câteva obiecte bazate pe clasa `Masina` pentru a ilustra acest concept, utilizând limbajele C++ și Python.

Exemplu 1.1 Definirea și utilizarea, în C++, a obiectului `Masina`

```
1  #include <iostream>
2  using namespace std;
3
4  class Masina {
5  private:
6      string marca;  // Atribut pentru marca masinii
7      string model;  // Atribut pentru modelul masinii
8      int an;        // Atribut pentru anul fabricatiei
9      int kilometraj; // Atribut pentru kilometraj
10
11 public:
12     Masina(string marca, string model, int an) {
13         this->marca = marca;
14         this->model = model;
15         this->an = an;
16         this->kilometraj = 0;
17     }
18
19     void conduce(int distanta) { // Metoda pentru a conduce masina
       ↪  si a actualiza kilometrajul
20         kilometraj += distanta;
21     }
22
23     void afiseazaInformatii() { // Metoda pentru a afisa
       ↪ informatiile despre masina
24         cout << "Marca: " << marca << ", Model: " << model << ",
```

```
            ↪ An: " << an << ", Kilometraj: " << kilometraj <<
            ↪ endl;
25      }
26  };
27
28  int main() {
29      Masina masina1("Toyota", "Camry", 2020); // Crearea unei
            ↪ instante a clasei Masina
30      Masina masina2("Honda", "Civic", 2018);
31
32      masina1.conduce(100); // Conducerea primei masini pe o
            ↪ distanta de 100 km
33      masina2.conduce(200); // Conducerea celei de-a doua masini pe
            ↪ o distanta de 200 km
34
35      masina1.afiseazaInformatii(); // Afisarea informatiilor despre
            ↪  prima masina
36      masina2.afiseazaInformatii(); // Afisarea informatiilor despre
            ↪  a doua masina
37
38      return 0;
39  }
```

La rularea exemplului 1.1, efectul afisat în consola este urmatorul:

```
Marca: Toyota, Model: Camry, An: 2020, Kilometraj: 100
Marca: Honda, Model: Civic, An: 2018, Kilometraj: 200
```

Exemplu 1.2 Definirea si utilizarea, în Python, a obiectului `Masina`

```
1  class Masina:
2      def __init__(self, marca, model, an):
3          # Atribut pentru marca masinii
4          self.marca = marca
5          # Atribut pentru modelul masinii
6          self.model = model
7          # Atribut pentru anul fabricatiei
8          self.an = an
9          # Atribut pentru kilometraj
10         self.kilometraj = 0
11
12     # Metoda pentru a conduce masina si a actualiza kilometrajul
13     def conduce(self, distanta):
```

```
14          self.kilometraj += distanta
15
16      # Metoda pentru a afisa informatiile despre masina
17      def afiseaza_informatii(self):
18          print(f"Marca: {self.marca}, Model: {self.model}, An: {
                ↪ self.an}, Kilometraj: {self.kilometraj}")
19
20  # Crearea instantelor clasei Masina
21  masina1 = Masina("Toyota", "Camry", 2020)
22  masina2 = Masina("Honda", "Civic", 2018)
23
24  # Conducerea masinilor pe distante diferite
25  masina1.conduce(100)
26  masina2.conduce(200)
27
28  # Afisarea informatiilor despre masini
29  masina1.afiseaza_informatii() # Output: Marca: Toyota, Model:
        ↪ Camry, An: 2020, Kilometraj: 100
30  masina2.afiseaza_informatii() # Output: Marca: Honda, Model: Civic
        ↪ , An: 2018, Kilometraj: 200
```

La rularea exemplului 1.2, efectul afisat în consola este urmatorul:

```
Marca: Toyota, Model: Camry, An: 2020, Kilometraj: 100
Marca: Honda, Model: Civic, An: 2018, Kilometraj: 200
```

Avantajele claselor și obiectelor în POO:

1. **Încapsularea:** Clasele permit încapsularea datelor și metodelor într-o unitate singulară, oferind o modalitate de control al accesului la date. Această încapsulare promovează ascunderea informațiilor, unde funcționarea internă a unui obiect poate fi păstrată privată față de exterior, reducând complexitatea și prevenind interferențele neintenționate.

2. **Modularitate și Reutilizabilitate:** Clasele facilitează modularitatea prin fragmentarea sistemelor complexe în unități mai mici și mai ușor de gestionat. Odată definite, clasele pot fi reutilizate în diferite părți ale codului sau în proiecte complet diferite, promovând reutilizarea codului și reducând redundanța.

3. **Moștenirea:** Moștenirea permite clase noi să fie derivate din cele existente, moștenindule atributele și comportamentele, permițând în același timp extinderea sau personalizarea acestora. Această structură ierarhică promovează organizarea codului și stimulează reutilizarea acestuia.

4. **Polimorfismul:** Polimorfismul permite obiectelor din clase diferite să fie tratate ca obiecte ale unei superclase comune, oferind flexibilitate și extensibilitate în cod. Acest lucru permite suprascrierea metodelor în subclase pentru a furniza implementări specifice, menținând totuși o interfață consistentă.

Dezavantajele claselor și obiectelor în POO:

1. **Complexitatea:** Sistemele orientate pe obiecte pot deveni complexe, în special cu ierarhiile de clase profunde și relațiile imbricate între obiecte. Gestionarea acestor complexități necesită un design și o implementare atentă, care uneori poate duce la suprasolicitare și scăderea performanței.

2. **Suprasolicitarea:** Programarea orientată pe obiecte poate introduce o suprasolicitare din cauza caracteristicilor precum dispecerizarea dinamică, alocarea de memorie și căutarea metodelor. Această suprasolicitare poate afecta performanța, în special în medii cu resurse limitate sau în aplicații care necesită un volum mare de procesare.

3. **Curba de învățare:** Înțelegerea și stăpânirea conceptelor POO, inclusiv a claselor și obiectelor, a moștenirii, polimorfismului și încapsulării, pot constitui o curba de învățare abruptă pentru începători. Caracterul abstract al acestor concepte poate necesita o schimbare în gândire pentru dezvoltatorii obișnuiți cu paradigmele de programare procedurală sau funcțională.

În concluzie, clasele și obiectele formează baza programării orientate pe obiecte, oferind numeroase avantaje precum încapsularea, modularitatea, moștenirea și polimorfismul. Cu toate acestea, ele vin și cu anumite dezavantaje, inclusiv complexitatea, suprasolicitarea și o curba de învățare. În cele din urmă, eficacitatea utilizării claselor și obiectelor depinde de cerințele specifice ale proiectului software și de abilitățile dezvoltatorilor implicați.

2. Fundamentele C++

Elemente de bază ale limbajelor C/C++
Structura programului C/C++
Directive de preprocesare
Utilizarea comentariilor
Tipuri de date fundamentale
Tipuri de date derivate
Tipuri de date definite de utilizator
Instrucțiuni
Funcții în C și C++

Limbajul C++ s-a dezvoltat la sfârşitul anilor '80, bazat fiind pe limbajul C care este un limbaj care folosește pe principiile programării procedurale, facilitând programarea structurată, cu posibilităţi de control, permiţând scrierea unor aplicaţii complexe, asemeni limbajelor de nivel înalt, dar în același timp se apropie şi de limbajele de nivel scăzut, permiţând inserarea de cod scris în limbaj de asamblare, ceea ce asigură o mare flexibilitate [5].

Odată cu dezvoltarea tehnicii de programare orientată pe obiecte POO (eng. OOP-Object Oriented Programming), Bjarne Stroustrup a creat limbajul "C cu clase", ulterior numit C++, care completează la limbajul C facilităţi de creare şi utilizare de clase [13]. Aplicarea principiilor POO duce la creşterea productivităţii programării.

Fundamentele limbajului C++ sunt preluate din limbajul C, el fiind standardizat prin adoptarea standardului ANSI C++ [3] [11].

Programele scrise în C++ sunt alcătuite din una sau mai multe funcţii (trebuie să conţină cel puţin o funcţie şi aceasta să se numească main). Nu este permisă crearea şi declararea unei funcţii în interiorul altei funcţii. Folosind variabile locale, se pot scrie proceduri care să realizeze o sarcină specifică şi care să nu cauzeze efecte secundare nedorite în alte zone ale codului [15] [4].

2.1 Elemente de bază ale limbajelor C/C++

La scrierea unui program în C/C++ se folosesc unităţi sintactice care se încadrează în diferite categorii, cum ar fi:

- **identificatori** - nume care desemnează tipuri de date, constante, variabile, funcţii, etc.
- **semne de punctuaţie şi caractere speciale** - simboluri care separă diverse entităţi ale programului, unii pot avea dublă semnificaţie;
- **spaţii albe** - sunt caractere cu rol de delimitare sau care cresc lizibilitatea programului sursă;
- **operatori** - simboluri utilizate pentru precizarea operaţiilor ce se efectuează;
- **constante** - valori fixe reprezentând valori numerice, întregi sau reale, caractere sau şiruri de caractere;
- **variabile** - entități care stochează valori ce pot fi modificate pe parcursul evoluției programului.

2.1.1 Identificatori

În limbajul C++, la fel ca în C standard, identificatorii sunt compuși din caractere alfanumerice și liniuța de subliniere (underscore). Aceștia sunt utilizați pentru a denumi tipuri de date, variabile, funcții, operatori, instrucțiuni etc. O cifră nu poate fi primul caracter al unui identificator. În continuare se pot observa o serie de exemple de identificatori:

```
1  Aria // valid
2  raza // valid
3  mesaj // valid
4  _maxx // valid
5  a5 // valid
6  5a // invalid, primul caracter este cifra
7  A_B_C // valid
8  A % B  // invalid, caracterul % nu poate fi folosit intr-un
       ↪ identificator
```

Limbajul C/C++ este **case sensitive**, adică face diferenţă între majuscule şi minuscule, astfel, în exemplele date anterior, numele Raza este diferit de raza. Limbajul are o serie de **cuvintele cheie (reserved keywords)**, adică identificatori cu o semnificație predefinită, care nu pot fi modificați de programator. Cuvintele cheie ale limbajului C++ sunt prezentate în Tabelul 2.1. Cele care încep cu caracterul **underscore** (_) reprezintă variabile interne.

2.1.2 Semne de punctuaţie şi caractere speciale

Semne de punctuaţie şi caractere speciale constituie un set de caractere cu diferite utilizări, care în funcţie de poziţia ocupată în textul programului sursă pot determina acţiunile care vor fi executate. Aceste caractere sunt: [ ] ( ) { } * , : = ; ... #.

Unele caractere pot desemna operatori (ca de exemplu [ ] * , =) altele nu desemnează o operaţie, ci delimitează zone în alcătuirea codului sursă. Spre exemplu, caracterul ; (punct şi virgulă) marchează sfârşitul unei instrucţiuni, iar { şi } (paranteze acolade) marchează începutul şi respectiv sfârşitul unui bloc de instrucţiuni. Caracterul # însoţeşte numai

Tabela 2.1 Cuvintele cheie ale limbajului C++.

Cuvinte cheie în C	**Cuvinte cheie în C++**
auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, inline, int, long, register, restrict, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while	alignas, alignof, asm, auto, bool, break, case, catch, char, class, const, constexpr, const_cast, continue, decltype, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, noexcept, nullptr, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_assert, static_cast, struct, switch, template, this, thread_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while

directivele de preprocesare. Semnificaţia şi modul de utilizare al acestor caractere se va preciza în capitolele următoare.

2.1.3 Spaţii albe

Spaţiile albe sunt caractere speciale care, deşi nu sunt afişabile, produc efecte în afişarea textului. Aceste caractere sunt: space (are valoarea 32 în codul ASCII), tab (are valoarea 9 în codul ASCII), newline (are valoarea 10 codul ASCII), vertical-tab (are valoarea 11 în codul ASCII), formfeed (are valoarea 12 în codul ASCII), carriage-return (are valoarea 13 în codul ASCII). Aceste caractere sunt interpretate ca separatori când sunt plasate între entităţi diferite ale programului sau ca şi caractere dacă intră în alcătuirea şirurilor de caractere. Sunt situaţii în care aceste caractere se folosesc pentru a creşte lizibilitatea programului sursă, caz în care ele vor fi ignorate de compilator.

2.1.4 Operatori în C și C++

Operatorii sunt simboluri utilizate pentru a specifica operațiile ce se efectuează asupra variabilelor și valorilor dintr-un program. Aceștia sunt esențiali în orice limbaj de programare, deoarece permit manipularea datelor și controlul fluxului execuției. În C și C++, operatorii sunt clasificați în mai multe categorii, în funcție de funcționalitatea lor.

Operatori aritmetici

Operatorii aritmetici sunt utilizați pentru efectuarea operațiilor matematice de bază. Aceștia includ:

- + – adunare

- `-` – scădere
- `*` – înmulțire
- `/` – împărțire
- `%` – restul împărțirii întregi (modulo)

Acești operatori pot fi utilizați cu tipuri de date numerice precum `int`, `float`, `double` și altele.

Operatori de atribuire

Operatorii de atribuire sunt folosiți pentru a asocia o valoare unei variabile. Cel mai simplu operator de atribuire este `=`, însă există și variante combinate care efectuează o operație și atribuie rezultatul în același timp:

- `=` – atribuie o valoare unei variabile
- `+=` – adaugă și atribuie rezultatul
- `-=` – scade și atribuie rezultatul
- `*=` – înmulțește și atribuie rezultatul
- `/=` – împarte și atribuie rezultatul
- `%=` – calculează restul împărțirii și atribuie rezultatul

Operatori relaționali

Operatorii relaționali sunt utilizați pentru a compara două valori și returnează un rezultat boolean (`true` sau `false`):

- `==` – verifică egalitatea a două valori
- `!=` – verifică inegalitatea
- `>` – verifică dacă prima valoare este mai mare decât a doua
- `<` – verifică dacă prima valoare este mai mică decât a doua
- `>=` – verifică dacă prima valoare este mai mare sau egală cu a doua
- `<=` – verifică dacă prima valoare este mai mică sau egală cu a doua

Operatori logici

Operatorii logici sunt folosiți pentru evaluarea expresiilor condiționale și combinarea rezultatelor boolean:

- `&&` – operatorul ȘI logic (returnează `true` doar dacă ambele condiții sunt adevărate)
- `||` – operatorul SAU logic (returnează `true` dacă cel puțin una dintre condiții este adevărată)
- `!` – operatorul de negare logică (inversează valoarea unei expresii)

Operatori pe biți

Acești operatori operează la nivel de biți și sunt folosiți pentru manipularea valorilor binare:

- `&` – operatorul ȘI bit cu bit
- `|` – operatorul SAU bit cu bit

- `^` – operatorul SAU exclusiv (XOR) bit cu bit
- `~` – operatorul de complement bit cu bit (inversare)
- `«` – operatorul de deplasare la stânga
- `»` – operatorul de deplasare la dreapta

Operatori de incrementare și decrementare

Acești operatori sunt folosiți pentru a crește sau a scădea valoarea unei variabile cu 1:

- `++` – incrementare (poate fi folosit înainte sau după variabilă: `++x` sau `x++`)
- `--` – decrementare (poate fi folosit înainte sau după variabilă: `-x` sau `x-`)

Operatori de acces și referință

Acești operatori sunt esențiali pentru lucrul cu pointeri și structuri:

- `*` – operatorul de dereferențiere (permite accesul la valoarea indicată de un pointer)
- `&` – operatorul de adresă (returnează adresa unei variabile)
- `->` – operatorul de acces la membrii unei structuri prin pointer
- `.` – operatorul de acces la membrii unei structuri sau clase

Operatori speciali

C și C++ includ și o serie de operatori cu funcționalități speciale:

- `sizeof` – returnează dimensiunea unui tip de date sau variabilă în octeți
- `typeid` – permite obținerea tipului unei variabile (în C++)
- `new` și `delete` – operatori de alocare și dealocare dinamică a memoriei (în C++)

Supraîncărcarea operatorilor în C++

Unul dintre avantajele majore ale C++ este posibilitatea de a supraîncărca operatorii, adică de a defini comportamente personalizate pentru tipuri de date definite de utilizator. Aceasta permite crearea unor clase care pot manipula obiecte într-un mod intuitiv folosind operatori standard.

Exemplu 2.1 Exemplu de supraîncărcare a operatorului `+`

```
1  #include <iostream>
2  class Vector {
3  public:
4      int x, y;
5      Vector(int a, int b) : x(a), y(b) {}
6
7      // Supraincarcarea operatorului +
8      Vector operator+(const Vector& v) {
9          return Vector(x + v.x, y + v.y);
10     }
11
```

```
12      void afiseaza() {
13          std::cout << "(" << x << ", " << y << ")" << std::endl;
14      }
15  };
16
17  int main() {
18      Vector v1(2, 3), v2(4, 5);
19      Vector v3 = v1 + v2;
20      v3.afiseaza(); // Output: (6, 8)
21      return 0;
22  }
```

La rularea exemplului 2.1, efectul afisat în consola este urmatorul:

```
(6, 8)
```

Prin utilizarea supraîncărcării operatorilor, programatorii pot crea clase mai intuitive și mai ușor de utilizat.

2.1.5 Constante

O constantă este o valoare care nu poate fi modificată după ce a fost atribuită. Aceasta este folosită pentru a preveni schimbările accidentale ale unor valori importante, ceea ce ajută la creșterea clarității și siguranței codului.

Definirea constantelor în C și C++

Există mai multe moduri de a defini constante în C și C++:

Utilizarea cuvântului cheie `const` Constantele pot fi declarate utilizând cuvântul cheie `const`, care indică faptul că valoarea variabilei nu poate fi modificată după inițializare.

Exemplu 2.2 Exemplu de utilizare a `const`

```
1  const double PI = 3.14159;
2  const int MAX_VALORI = 100;
```

Utilizarea directivei de preprocesor `#define` În C și C++, directivele preprocesorului pot fi utilizate pentru a defini constante simbolice. Această metodă este mai veche și nu oferă verificare de tip.

Exemplu 2.3 Exemplu de utilizare a `#define`

```
1  #define MAX_VITEZA 120
2  #define MESAJ "Bun venit!"
```

Utilizarea `constexpr` în C++ C++ introduce cuvântul cheie `constexpr`, care definește constante ce pot fi evaluate la compilare. Acest lucru permite optimizări suplimentare.

Exemplu 2.4 Exemplu de utilizare a `constexpr`

```
1 constexpr int DUBLU(int x) { return x * 2; }
2 constexpr double G = 9.81;
```

Diferențe între `const`, `#define` și `constexpr`

- `const` – permite definirea constantelor cu verificare de tip, dar valoarea este stabilită la execuție.
- `#define` – folosit pentru substituții textuale, dar fără verificare de tip.
- `constexpr` – permite evaluarea la compilare, îmbunătățind performanța și siguranța.

2.1.6 Variabile

O variabilă este un nume simbolic asociat unei zone de memorie, care poate stoca o valoare de un anumit tip de date. Variabilele pot fi modificate în timpul execuției programului, oferind flexibilitate în procesarea informației.

Declararea și inițializarea variabilelor

În C și C++, o variabilă trebuie declarată înainte de a fi utilizată. Declarația specifică tipul de date și numele variabilei, iar opțional, aceasta poate fi inițializată cu o valoare inițială.

Exemplu 2.5 Exemple de declarare și inițializare a variabilelor

```
1 int numar;          // Declararea unei variabile de tip intreg
2 double pi = 3.14;   // Declarare si initializare
3 char litera = 'A'; // Variabila de tip caracter
```

Tipuri de variabile

Variabilele pot fi de mai multe tipuri, în funcție de domeniul lor de aplicabilitate și durata de viață:

- **Variabile locale** – declarate în interiorul unei funcții sau bloc de cod și accesibile doar în acel context.
- **Variabile globale** – declarate în afara funcțiilor și accesibile în întregul program.
- **Variabile statice** – păstrează valoarea între apeluri succesive ale funcției în care sunt declarate.
- **Variabile volatile** – marcate cu `volatile`, indicând compilatorului că valoarea lor se poate schimba în mod neașteptat (ex. în cazul interacțiunii cu hardware-ul).

Modificatori de acces și de stocare

C și C++ oferă modificatori care controlează comportamentul variabilelor:

- `auto` – specifică că tipul variabilei este dedus automat (în C++11+).
- `static` – păstrează valoarea variabilei între apelurile funcției.
- `extern` – permite accesul la o variabilă definită într-un alt fișier sursă.
- `register` – sugerează compilatorului utilizarea unui registru hardware (mai puțin utilizat în prezent).

Utilizarea corectă a variabilelor și constantelor este esențială pentru un cod clar, eficient și ușor de întreținut. Constantele protejează valorile critice împotriva modificărilor accidentale, iar variabilele permit flexibilitatea necesară în manipularea datelor. Adoptarea celor mai bune practici în gestionarea acestora contribuie la scrierea unui cod robust și eficient.

2.2 Structura programului C/C++

Acţiunile efectuate de un program C/C++ se pot grupa respectând algoritmul transpus, programul fiind structurat în module, fiecare având un rol bine definit. Fiecare acţiune este determinată de o instrucţiune. O succesiune de instrucţiuni poate fi grupată într-un bloc numit funcţie. Structura generală a unui program este alcătuită, în general, din 3 zone:

```
1 I   < directive de preprocesare >
2 II  < declartii globale >
3 III definitii de functii
```

Zonele I şi II sunt opţionale, dar cea de a III-a este obligatoriu să existe.

Preprocesorul efectuează operaţii înainte de compilarea codului, indicate prin aşa numite directive de preprocesare, marcate totdeauna de caracterul #. Ele produc modificări în codul care va fi compilat, cum ar fi inserarea unor fişiere fie din biblioteca standard, fie create de utilizator, declararea de macrodefiniţii, etc. **Declaraţiile globale** pot conţine definiţii de tipuri de date, prototipuri de funcţii, declaraţii de variabile globale.

Programul trebuie să conţină cel puţin o funcţie, denumită **main()**, execuţia programului începând cu această funcţie. Funcţia main() este unică, ea nu poate fi supradefinită. Restul funcţiilor pot efectua diferite acţiuni, ele putând fi definite de utilizator, sau să fie preluate din bibliotecile de funcţii C++.

Funcţiile în C++, spre deosebire de alte limbaje, cum ar fi Pascal, nu pot include definirea altor funcţii. Deci funcţiile sunt definite în afara oricărei funcţii. Definiţia unei funcţii are următoarea structură:

```
1 <tip_r> nume_functie (< lista_parametri >)
2 {
3     <declaratii locale >
4     secventa de instructiuni
5 }
```

Unde: `tip_r` - precizează tipul rezultatului funcţiei. Dacă nu este specificat, în mod implicit rezultatul întors de funcţie este o valoare numerică întreagă, tipul de dată fiind int. Dacă funcţia nu returnează nici o valoare, acest lucru este precizat prin tipul de dată

void. Dacă funcţia prelucrează date existente în momentul apelului funcţiei (date de intrare), aceste date pot fi preluate prin lista de parametri (lista_parametri). La apelul funcţiei este obligatoriu să se transmită valori efective pentru fiecare parametru. Programul sursă poate fi alcătuit din una sau mai multe funcţii. Acestea pot fi definite într-un singur fişier sursă, sau în mai multe fişiere sursă, acestea putând fi compilate separat, legăturile între acestea fiind realizate în faza de link-editare pentru a genera un singur fişier executabil.

2.3 Directive de preprocesare

Preprocesorul efectuează operaţii prealabile compilării, indicate de directivele de preprocesare. Acestea sunt precedate de caracterul #. În mod frecvent sunt utilizate directivele #define şi #include.

Directiva # define

Directiva define este utilizată pentru a asocia un identificator cu un şir de caractere. În etapa de precompilare fiecare apariţie a identificatorului va fi înlocuită, caracter cu caracter, cu şirul corespunzător. Forma generală de utilizare a directivei #define este: `#define nume sir_ de_caractere`

Exemple de definiţii:

```
1 #define TRUE    1
2 #define FALSE   0
3 #define NULL    0
4 #define MIN     10
5 #define MAX   MIN+100
6 #define  f(a,b)   a+b
```

Directiva #define :
- permite folosirea constantelor simbolice în programe
- uzual, dar nu obligatoriu, simbolurile sunt scrise cu litere mari
- nu se termină cu ';'
- apar în general la începutul unui fişier sursă
- fiecare apariţie a simbolului este înlocuită în etapa de precompilare, caracter cu caracter, cu valoarea sa.

Directiva #include

Directiva de preprocesare #include determină includerea unui fişier sursă în alt fişier sursă. Sintaxa de utilizare are una din formele:

```
1 #include  <nume_fisier>
2 #include "nume_fisier"
```

Pentru prima variantă, fişierul va fi căutat într-un director predefinit care conţine fişiere

header puse la dispoziţie de mediul de programare, cea de a doua va căuta fişierul pe calea specificată în numele fişierului. Dacă nu e precizată calea fişierului, va fi căutat în directorul curent.

Exemple de folosire a directivei #include:

```
1 #include <stdio.h> // fisierul e cautat in directorul predefinit
2 #include "c:\dir\fis_meu1.h" // fisierul este cautat pe calea c:\
    ↪ dir
3 #include "fis_meu2.h" // fisierul este cautat in directorul curent
```

2.4 Utilizarea comentariilor

La scrierea programelor sursă este util în numeroase situaţii să se introducă anumite comentarii care să explice eventuale secvenţe de instrucţiuni. Comentariile sunt ignorate de compilator putând conţine orice text, ele fiind utile doar programatorului. Comentariile se pot scrie astfel:

- comentarii pe un rând – sunt precedate de caracterele //
- comentarii pe mai multe rânduri – sunt marcate între caracterele /* şi */

2.5 Tipuri de date fundamentale

Tipurile de date fundamentale definesc natura valorilor pe care variabilele le pot stoca și modul în care acestea sunt gestionate în memorie. În C și C++, aceste tipuri sunt esențiale pentru gestionarea eficientă a resurselor și pentru scrierea unui cod performant.

2.5.1 Clasificarea tipurilor fundamentale de date

Tipurile fundamentale de date în C și C++ pot fi clasificate astfel:

- **Tipuri întregi** – numere întregi cu sau fără semn.
- **Tipuri reale (în virgulă mobilă)** – numere zecimale pentru calcule cu precizie variabilă.
- **Tipul caracter** – stocarea caracterelor individuale în format ASCII sau Unicode.
- **Tipul boolean** – valori logice `true` sau `false`.
- **Tipul vid** – indică absența unui tip de date, utilizat pentru funcții fără valoare de retur.

2.5.2 Tipuri întregi (`int`, `short`, `long`)

Tipurile întregi sunt utilizate pentru a stoca valori numerice fără componente fracționare. Dimensiunea acestora depinde de arhitectura sistemului (32-bit sau 64-bit) și de compilator.

Memoria este alocată liniar, fiecare variabilă ocupând un număr fix de octeți în RAM. Pe sisteme pe 32 de biți, un `int` ocupă 4 octeți, iar pe sisteme pe 64 de biți poate ocupa 8 octeți.

Utilizarea semnului: tipurile întregi pot fi **cu semn** (signed) sau **fără semn** (unsigned):

Tip	Dimensiune tipică (bytes)	Interval de valori
`short int`	2	-32,768 până la 32,767
`int`	4	-2,147,483,648 până la 2,147,483,647
`long int`	4 sau 8	Depinde de platformă
`long long int`	8	Aproximativ -9×10^{18} până la 9×10^{18}

Tabela 2.2 Dimensiuni tipice pentru tipurile întregi

Exemplu 2.6 Exemplu de variabile cu semn și fără semn

```
signed int x = -10;  // Poate stoca valori negative
unsigned int y = 250; // Doar valori pozitive
```

Un `unsigned int` extinde intervalul valorilor pozitive, dar nu poate stoca valori negative.

2.5.3 Tipuri reale (`float`, `double`, `long double`)

Tipurile reale sunt utilizate pentru a stoca valori numerice cu virgulă mobilă. Precizia lor depinde de numărul de biți folosiți pentru reprezentarea mantisei și exponentei.

Tip	Dimensiune (bytes)	Precizie aproximativă
`float`	4	6-7 zecimale
`double`	8	15-16 zecimale
`long double`	8-16	Până la 19 zecimale (depinde de sistem)

Tabela 2.3 Dimensiuni și precizie pentru tipurile reale

Reprezentarea în memorie: numerele în virgulă mobilă sunt stocate conform standardului IEEE 754, folosind 3 componente:

- **Semn** – 1 bit care indică pozitiv (`0`) sau negativ (`1`).
- **Exponent** – reglează poziția virgulei mobile.
- **Mantisa** – stochează cifrele semnificative.

2.5.4 Tipul caracter (`char`)

Tipul `char` este utilizat pentru a stoca caractere individuale, fiind reprezentat intern ca un număr între 0 și 255 (cod ASCII).

Exemplu 2.7 Exemplu de utilizare a tipului `char`

```
char litera = 'A';    // Caracter ASCII
char simbol = 65;     // Echivalent cu 'A'
```

În memorie, un caracter ocupă 1 octet. Pentru seturi de caractere extinse, C++ oferă `wchar_t` (pentru Unicode).

2.5.5 Tipul boolean (`bool`)

Introducerea tipului `bool` în C++ permite gestionarea logică a valorilor `true` și `false`:

Exemplu 2.8 Exemplu de utilizare a tipului `bool`

```
1 bool esteAdevarat = true;
2 bool esteFals = false;
```

Deși ocupă teoretic 1 bit, compilatoarele alocă de obicei 1 octet (8 biți) pentru eficiență.

2.5.6 Tipul vid (`void`)

Tipul `void` indică absența unui tip de date și este utilizat pentru funcții care nu returnează o valoare:

Exemplu 2.9 Exemplu de utilizare a tipului `void`

```
1 void salut() {
2     std::cout << "Salut!" << std::endl;
3 }
```

De asemenea, este utilizat pentru declarații de pointeri generici:

Exemplu 2.10 Utilizare pointer `void *`

```
1 void *ptr; // Pointer generic
```

2.5.7 Alinierea și eficiența memoriei

În arhitectura modernă, variabilele sunt aliniate în memorie pentru acces mai rapid. De exemplu, un `int` de 4 octeți va fi plasat la o adresă multiplă de 4, iar un `double` de 8 octeți la o adresă multiplă de 8.

Exemplu 2.11 Alinierea variabilelor în memorie

```
1 struct Exemplu {
2     char c;       // 1 byte
3     int x;        // 4 bytes, dar este plasat pe urmatorul multiplu
      ↪ de 4
4     double d;     // 8 bytes, aliniat pe 8
5 };
```

Această aliniere poate cauza **padding** (goluri de memorie) între variabile pentru a respecta regulile de aliniere. Tipurile fundamentale de date în C și C++ sunt esențiale pentru manipularea eficientă a memoriei. Înțelegerea reprezentării lor la nivel de memorie ajută la optimizarea performanței și la scrierea unui cod mai robust.

2.6 Tipuri de date derivate

2.6.1 Pointeri de date

Un pointer este o dată a cărei valoare este o adresă de memorie. O variabilă de tip pointer este o variabilă a cărei valoare este adresa altor variabile.

Sintaxa declaratiei unui pointer de date este:

```
tip * id_ptr;
```

Simbolul * indică faptul că `id_ptr` este o variabilă de tip pointer, iar `tip` reprezintă tipul datelor către care pointerul face referire (adică tipul de bază al lui `id_ptr`).

La compilare, adresa memorată în pointer este tratată de compilator ca o locație ce conține un obiect de tipul specificat, având toate caracteristicile asociate acelui tip: dimensiunea corespunzătoare în memorie și modul de interpretare a datelor stocate.

```
int * ptr; // pointer la int
int * tabptr[10]; // tablou de pointeri catre int
float **pptr; // dubla indirectare; pointer la pointer
```

Doi operatori unari sunt esențiali pentru utilizarea pointerilor: operatorul `&` permite obținerea adresei unei variabile, iar operatorul * este folosit pentru a accesa valoarea stocată la acea adresă prin intermediul pointerului.

Este important ca adresa stocată într-un pointer să corespundă tipului specificat în declarația sa.

```
float variabila;
float*pointer = &variabila;  // declararea unui pointer la void si
    ↪ initializarea lui cu adresa
                 // variabilei variabila
*pointer = 3.5; // se atribuie valoare variabilei de la adresa
    ↪ referita prin pointer
```

Orice variabilă de tip pointer trebuie inițializată cu o valoare validă, cum ar fi 0 (NULL) sau adresa unei variabile existente, înainte de a fi folosită. În absența unei astfel de inițializări, pot apărea consecințe serioase, deoarece pointerii oferă acces direct la date, care pot fi modificate neintenționat. Nici compilatorul și nici timpul de execuție nu verifică dacă valorile pointerilor sunt valide.

Cu variabile pointer se pot efectua operatii aritmetice de adunare (+) si de scadere (-) si operatiile unare de incrementare (++) si de decrementare (–).

Interpretarea acestor operatii este următoarea: avand variabila pointer tip * ptr, expresia ptr + n returnează ca valoare adresa memorată în ptr la care se adauga n * sizeof(tip), de exemplu:

```
float variabila;
float*pointer = &variabila;
```

Expresia `pointer + 2` returnează ca valoare adresa memorată în pointer, la care se adaugă 2*sizeof(float) octeți, adică 8 octeti.

O categorie distinctă de pointeri o constituie pointerii `void`, numiți și pointeri generici, care pot conține adresa obiectelor de orice tip.

Declararea unui pointer void se face folosind următoarea sintaxă:

```
1 void * id_vptr;
```

În cazul unui pointer de tip `void`, nu sunt definite nici dimensiunea memoriei la care face referire, nici modul de interpretare a conținutului acelei memorii, iar comportamentul acestuia diferă față de pointerii către tipuri specifice.

Pentru a accesa obiectul indicat de un pointer `void`, este obligatorie convertirea explicită a acestuia către un tip concret, folosind operatorul de cast.

Variabilele pointer sunt folosite în mod uzual în asociere cu tablouri de date (i), folosindu-se aritmetica cu pointeri, pentru alocări dinamice de memorie (ii) sau în transferul de date între funcții (iii).

Alocarea de memorie dinamică se realizeazaă în mod explicit, cu ajutorul funcțiilor de alocare dinamică definite în fișierele header `<stdlib.h>` si `<alloc.h>` (de exemplu, funcțiile malloc, calloc, realloc pentru alocare, respectiv free pentru eliberarea memoriei), sau cu operatorul unar `new`, după cum urmează:

```
1 var_ptr = new tip_var; // 1
2 var_ptr = new tip_var(val_init); // 2
3 var_ptr = new tip_var[dim]; // 3
```

unde: `var_ptr` este o variabila pointer de un tip oarecare, `tip_var` este tipul variabilei dinamice `var_ptr`, respectiv `val_init` este expresie cu a cărei valoare se inițializează variabila dinamică.

În cazul alocarii de memorie cu operatorul new, eliberarea memoriei se realizează folosind operatorul `delete` după cum urmează:

```
1 delete var_ptr; // in cazul alocarii cu 1 sau 2
2 delete[] var_ptr; // in cazul alocarii cu 3
```

2.6.2 Variabile referintă

Limbajul C++ permite definirea unor identificatori care acționează ca referințe către obiecte (fie variabile, fie constante). Asemenea pointerilor, referințele stochează adrese. Declararea unei referințe se realizează cu ajutorul simbolului `&`, utilizând următoarea sintaxă:

```
1 tip & id_referinta = nume_obiect;
```

unde, `tip` reprezintă tipul obiectului la care se face referire, simbolul `&` indică faptul că `id_referinta` este o variabilă referință, iar `nume_obiect` este obiectul a cărui adresă va fi asociată acestei referințe.

```
1 int n; // se declara n de tip int
2 int &r=n; // se defineste r referinta lui n
```

Referințele trebuie întotdeauna inițializate în momentul declarării și nu pot fi reasociate ulterior; ele funcționează ca un alias pentru variabila a cărei adresă o dețin. Nu este permisă

declararea de referințe către câmpuri de biți, referințe la referințe sau pointeri la referințe, prin urmare nu se pot crea nici tablouri de referințe. Deși rareori sunt utilizate ca entități independente, referințele sunt frecvent folosite pentru transmiterea datelor între funcții.

2.6.3 Tablouri de date

Tablourile de date sunt colectii de date de acelasi tip, plasate în zone contigue de memorie. Declarația unui tablou cu N dimensiuni se realizează astfel:

```
1 tip_element nume_tablou [dim1][dim2]...[dimN];
```

În memorie, zona ocupată are dimensiunea:

$$dim1 \times dim2 \times \ldots \times dimN$$

elemente de tipul `tip_element`.

Pentru referirea unui element de tablou se utilizează peratorul de indexare [], după cum urmează:

```
1 nume_tablou [indice1][indice2]...[indiceN];
```

Declararea unui tablou cu valori inițiale se face după cum urmează:

```
1 tip_element nume_tablou [dim1][dim2]...[dimN] = {lista_valori};
```

Valorile din listă trebuie să fie constante compatibile cu tipul de bază al tabloului și să fie specificate în ordinea în care acestea vor fi stocate în memorie.

```
1 float vect1[0]; // se declara un tablou cu 5 elemente float, fara
    ↪ initializare
2 vect1[0]=2.5; // elementului de index 0 i se atribuie valoarea 2.5
3 int vect2[10]={1,2,3,4,5,6,7,8,9,10}; // se declara un tablou cu
    ↪ 10 elemente int cu initializare
4 mat[1][2]=16; // elementului de indici 1 si 2 i se atribuie
    ↪ valoarea 16
5 int mat[2][3]={{1,2,3},{4,5,6}}; // se declara un tablou
    ↪ bidimensional cu 2*3 elemente int
```

Tablourile unidimensionale de tip `char` sunt utilizate pentru stocarea șirurilor de caractere. Sfârșitul unui șir este semnalat prin adăugarea unui octet cu valoarea 0 (`\0`) imediat după ultimul caracter, acesta fiind cunoscut sub denumirea de terminator de șir.

```
1 char sir1[10]; // se declara un tablou unidimensional de char,
    ↪ fara initializare
2 char sir2[] = "mai multe caractere"; // se declara un tablou de 20
    ↪  caractere, cu initializare
3  // (ultimul caracter este '\0')
```

```
4 sir2[0] = 'M'; // primul caracter al sirului primeste valoarea 'M'
```

Atunci când este utilizat fără index, numele unui tablou acționează ca un pointer constant către tipul elementelor sale și indică adresa primului element. Accesul la elementele tabloului se poate realiza și prin operații aritmetice aplicate asupra acestui pointer.

```
1 float tab[20], *ptr;
2 ptr = tab; // atribuire valida, ptr contine adresa primului
    ↪ element
3 &tab[0] == tab; // expresie adevarata
4 &tab[1] == tab + 1; // expresie adevarata
5 tab[0] == *tab; // expresie adevarata
6 tab[1] == *(tab + 1); // expresie adevarata
7 *ptr == tab[0]; // expresie adevarata
8 tab++; // eroare - pointer constant
9 ptr++; // corect
```

Tablourile multidimensionale sunt structuri în care fiecare element este, la rândul său, un tablou, iar numele tabloului (utilizat fără indexare) este interpretat ca un pointer către un tablou.

```
1 float mat[5][5]; // mat reprezinta un tablou de pointeri float
2 float *p;
3 p = mat; // eroare - tipuri diferite
4 p = (float*)mat; // corect - conversie explicita
5 mat == &mat[0][0]; // expresie adevarata
6 mat + 1 == &mat[1][0]; // expresie adevarata
7 *(mat + 4) == mat[4][0]; // expresie adevarata
```

2.7 Tipuri de date definite de utilizator

Tipurile de date fundamentale și cele derivate nu sunt întotdeauna suficiente pentru a satisface cerințele de structurare a informației, fiind adesea necesară combinarea mai multor tipuri de date diferite într-o singură entitate.

2.7.1 Enumerarea

Un tip enumerare este definit ca un set de constante întregi, fiecare legată de un identificator specific. Declararea unei enumerări se face după cum urmează:

```
1 enum <id_tip_enum> {id_elem <= const>, ...} <lista_id_var>;
```

unde, `id_tip_enum` desemnează numele tipului enumerat, `id_elem` indică numele fiecărui element, iar `const` reprezintă valoarea constantă atribuită acelui element.

În absența unor valori specificate explicit, compilatorul atribuie implicit valoarea 0 primului element, iar următoarele elemente vor avea valori incrementate automat cu câte 1 față

de precedentul.

Toți identificatorii dintr-o enumerare trebuie să fie unici în cadrul contextului în care sunt declarați, neputând avea același nume cu alte variabile, funcții sau tipuri definite prin `typedef`.

Utilizarea tipului enumerare este avantajoasă atunci când o variabilă poate primi doar un set restrâns de valori întregi, fiecare cu o denumire sugestivă. Astfel, codul devine mai lizibil și mai ușor de întreținut.

2.7.2 Structuri de date

O structură reprezintă un grup de date organizate sub un singur nume. Printr-o declarație de structură se specifică identificatorii și tipurile fiecărui element component, definind astfel un nou tip de date personalizat.

Sintaxa declaratiei unui tip structura este:

```
1 struct id_tip_struct {
2     tip_elem1 id_elem1;
3     tip_elem2 id_elem2;
4     ...
5     tip_elemN id_elemN;
6 } lista_id_var_struct;
```

unde: `struct` este cuvântul cheie folosit pentru definirea unui tip de structură, `id_tip_struct` este numele tipului structurat declarat, `tip_elemK` indică tipul celui de-al K-lea element (unde K variază de la 1 la N), `id_elemK` este numele acelui element, iar `lista_id_var_struct` conține numele variabilelor care vor avea tipul definit prin structură [18].

Într-o astfel de declarație, este permis să lipsească fie numele structurii, fie lista variabilelor declarate, însă nu ambele simultan. În general, se recomandă specificarea numelui structurii pentru a permite reutilizarea ulterioară a tipului în alte declarații.

Componentele unei structuri sunt denumite în mod generic membri (sau câmpuri). Tipul fiecărui membru poate fi oricare, cu excepția structurii care se definește, însă este permisă utilizarea de pointeri către acel tip de structură.

O variabilă de tip structură poate fi inițializată specificând valorile membrilor în exact aceeași ordine în care aceștia sunt declarați.

Accesul la un membru al unei variabile de tip structură se realizează cu operatorul de selecție . (punct), astfel:

```
1 nume_variabila.nume_camp;
```

Atunci când o structură este accesată prin intermediul unui pointer, membrii săi se accesează utilizând operatorul de selecție indirectă -> (săgeată), după cum urmează:

```
1 nume_variabila->nume_camp;
```

Operatorul de atribuire poate fi folosit între variabile de tip structură, cu condiția ca

acestea să fie de același tip, iar atribuirea se realizează automat pentru fiecare membru în parte.

Cu date de tip structura se pot folosi de asemenea operatorul adresa (`&`), `sizeof`, `*` (unar-pointer), in timp ce operatii aritmetice, relationale, logice, etc. nu pot fi efectuate cu astfel de date.

2.7.3 Câmpuri de biți

Câmpul de biți este un membru al unei structuri sau uniuni care are alocat un grup de biți în interiorul unui cuvant de memorie.

O structură cu membri câmpuri de biți se declară sub forma:

```
1 struct id_tip_struct {
2     tip_elem1 id_elem1 : lungime;
3     tip_elem2 id_elem2 : lungime;
4     ...
5     tip_elemN id_elemN : lungime;
6 } lista_id_var_struct;
```

unde, `lungime` indică numărul de biți alocați în memorie pentru reprezentarea câmpului respectiv.

La declararea câmpurilor de biți trebuie respectate următoarele restricții:

- tipul permis este `int`, fie semnat (`signed`), fie nesemnat (`unsigned`)
- valoarea `lungime` trebuie să fie o constantă întreagă cuprinsă între 0 și 15
- nu se poate obține adresa unui câmp de biți, deci operatorul `&` nu este aplicabil
- nu este permisă utilizarea de tablouri formate din câmpuri de biți

Accesul la membrii structurii care conține câmpuri de biți se realizează în același mod ca la structurile obișnuite, utilizând operatorii de selecție directă (`.`) sau indirectă (`->`).

2.7.4 Uniuni de date

O uniune oferă posibilitatea ca mai multe obiecte, de tipuri diferite, să partajeze aceeași zonă de memorie.

Declarația unei uniuni urmează o sintaxă asemănătoare cu cea utilizată pentru structuri:

```
1 union id_tip_uniune {
2     tip_elem1 id_elem1;
3     tip_elem2 id_elem2;
4     ...
5     tip_elemN id_elemN;
6 } lista_id_var_uniune;
```

Memoria alocată pentru o uniune este determinată de dimensiunea celui mai mare tip de date dintre membrii săi, iar toți membrii partajează aceeași locație de memorie.

2.8 Instructiuni

O instrucțiune în C/C++ este o expresie care se termină cu caracterul (;). In continuare sunt prezentate instrucțiunile în C/C++.

Instrucțiunea vidă

Prototip:

```
1 // Instructiune vida
2 ;
```

Această instrucțiune nu face nimic; este adesea folosită pentru a marca un loc valid unde o instrucțiune este cerută sintactic, dar nu este necesară logic.

Instrucțiunea compusă (blocul de instrucțiuni)

Prototip:

```
1 {
2     // lista_declaratii;
3     // lista_instructiuni;
4 }
```

Exemplu:

```
1 {
2     int a = 2, b = 5;
3     printf("\n%d + %d = %d", a, b, a + b);
4     printf("\n%d x %d = %d", a, b, a * b);
5 }
```

Un bloc de instrucțiuni este sintactic echivalent cu o singură instrucțiune.

Instrucțiunea `if` și `if-else`

Prototip:

```
1 if (expresie)
2     instructiune_a;
3
4 if (expresie)
5     instructiune_a;
6 else
7     instructiune_b;
```

Exemplu:

```
1 int a;
2 ...
```

```
3 if (a % 2 == 0)
4     printf("\na = %d este valoare para", a);
5 else
6     printf("\na = %d este valoare impara", a);
```

`if` evaluează expresia; dacă este adevărată (nenulă), se execută prima instrucțiune; altfel, se execută cea de-a doua (dacă există).

Instrucțiunea `while`

Prototip:

```
1 while (expresie)
2     instructiune;
```

Exemplu:

```
1 int a = 0;
2 ...
3 while (a <= 10)
4     printf("\na = %d", a++);
```

Execuția instrucțiunii se repetă atâta timp cât expresia este adevărată.

Instrucțiunea `do-while`

Prototip:

```
1 do
2     instructiune;
3 while (expresie);
```

Exemplu:

```
1 int a = 0;
2 ...
3 do {
4     printf("\na = %d", a++);
5 } while (a <= 10);
```

Instrucțiunea este executată cel puțin o dată; condiția este verificată la final.

Instrucțiunea `for`

Prototip:

```
1 for (expresie1; expresie2; expresie3)
2     instructiune;
```

Exemplu:

```
int a;
...
for (a = 0; a <= 10; a++)
    printf("\na = %d", a);
```

Această instrucțiune este des folosită pentru iterații controlate.

Instrucțiunea `switch`

Prototip:

```
switch (expresie) {
    case const_1: instructiuni; break;
    case const_2: instructiuni; break;
    ...
    default: lista_instructiuni;
}
```

Exemplu:

```
char c;
...
c = getchar();
switch (c) {
    case '0':
    case '1':
    case '2':
    case '3':
    ...
    case '9':
        printf("Tasta apasata este o cifra!");
        break;
    default:
        printf("Tasta apasata nu este o cifra!");
}
```

`switch` permite selecția între mai multe ramuri în funcție de o valoare întreagă.

Instrucțiunea `break`

Prototip:

```
break;
```

Exemplu:

```
for (int a = 0; ; a++) {
```

```
2       printf("\na = %d", a);
3       if (a == 10)
4           break;
5   }
```

`break` întrerupe execuția unui ciclu sau a unei instrucțiuni `switch`.

Instrucțiunea `continue`

Prototip:

```
1   continue;
```

Exemplu:

```
1   for (int i = 1; i < 100; i++) {
2       printf("\ni = %d", i);
3       if (!(i % 7)) {
4           printf("\ni = %d - divizibil cu 7", i);
5           continue;
6       }
7   }
```

`continue` trece direct la următoarea iterație a buclei.

Instrucțiunea `return`

Prototipuri:

```
1   return;
2   return expresie;
```

Exemplu 1:

```
1   void mesaj(void) {
2       printf("Se afiseaza un mesaj");
3       return;
4   }
```

Exemplu 2:

```
1   float arie_cerc(float r) {
2       float arie = PI * patrat(r);
3       return arie;
4   }
```

`return` oprește execuția funcției și poate returna o valoare.

Instrucțiunea `goto`

Prototip:

```
goto eticheta;

eticheta:
    instructiune;
```

Exemplu:

```
int a = 1;
...
et1:
printf("\na = %d", a);
a++;
if (a < 10)
    goto et1;
```

Saltul necondiționat este rar folosit în practică, deoarece reduce lizibilitatea codului.

2.9 Funcții în C și C++

Funcțiile reprezintă blocuri de cod reutilizabile, care îndeplinesc o anumită sarcină și pot fi apelate ori de câte ori este necesar. Ele contribuie la modularizarea codului, îmbunătățind claritatea, întreținerea și reutilizarea acestuia.

2.9.1 Definiția și apelul funcțiilor

O funcție este definită prin specificarea tipului de retur, a numelui și a listei de parametri. Sintaxa generală este următoarea:

Exemplu 2.12 Definiția unei funcții

```
tip_de_retur nume_functie(lista_parametri) {
    // Corpul functiei
    return valoare; // (optional, doar daca tipul de retur nu este
       ↪ void)
}
```

Funcțiile sunt apelate prin specificarea numelui și transmiterea argumentelor necesare:

Exemplu 2.13 Apelarea unei funcții

```
int suma(int a, int b) {
    return a + b;
}

int main() {
```

```
6       int rezultat = suma(5, 10); // Apelul functiei
7       return 0;
8   }
```

2.9.2 Tipuri de funcții

Funcțiile pot fi clasificate în funcție de comportamentul lor:

- **Funcții cu return** – returnează o valoare după execuție.
- **Funcții fără return** (`void`) – efectuează o acțiune, dar nu returnează o valoare.
- **Funcții cu parametri** – primesc argumente pentru a efectua operații specifice.
- **Funcții fără parametri** – nu necesită date de intrare.
- **Funcții recursive** – se apelează pe ele însele pentru a rezolva probleme iterative.
- **Funcții inline** – sugerează compilatorului să înlocuiască apelul funcției cu codul acesteia pentru optimizare.
- **Funcții supraincarcate** – în C++, permite definirea mai multor funcții cu același nume, dar parametri diferiți.

Funcții cu return

Aceste funcții returnează o valoare după execuție. Tipul de retur este specificat în definiția funcției, iar rezultatul este transmis prin instrucțiunea `return`.

Exemplu 2.14 Exemplu de funcție cu return

```
1   int suma(int a, int b) {
2       return a + b; // Returneaza suma celor doua numere
3   }
4
5   int main() {
6       int rezultat = suma(5, 10);
7       std::cout << "Suma este: " << rezultat;
8   }
```

Funcții fără return (`void`)

Aceste funcții execută acțiuni, dar nu returnează o valoare. Se folosesc în principal pentru operații care nu necesită un rezultat.

Exemplu 2.15 Exemplu de funcție fără return

```
1   void afiseazaMesaj() {
2       std::cout << "Salut, lume!" << std::endl;
3   }
4
5   int main() {
6       afiseazaMesaj(); // Apelul functiei
```

```
7 }
```

Funcții cu parametri

O funcție poate primi date de intrare sub forma parametrilor. Aceștia sunt specificați în lista de parametri și pot fi utilizați în corpul funcției.

Exemplu 2.16 Exemplu de funcție cu parametri

```
1 void salutare(std::string nume) {
2     std::cout << "Salut, " << nume << "!" << std::endl;
3 }
4
5 int main() {
6     salutare("Alex");
7 }
```

Funcții fără parametri

Aceste funcții nu necesită date de intrare și sunt utilizate pentru acțiuni generale.

Exemplu 2.17 Exemplu de funcție fără parametri

```
1 int obtineNumarAleator() {
2     return rand() % 100; // Returneaza un numar intre 0 si 99
3 }
4
5 int main() {
6     int numar = obtineNumarAleator();
7     std::cout << "Numar generat: " << numar;
8 }
```

Funcții recursive

Funcțiile recursive se apelează pe ele însele pentru a rezolva probleme iterative, cum ar fi calculul factorialului sau al seriilor Fibonacci.

Exemplu 2.18 Exemplu de funcție recursivă pentru factorial

```
1 int factorial(int n) {
2     if (n == 0) return 1; // Cazul de baza
3     return n * factorial(n - 1); // Apel recursiv
4 }
5
6 int main() {
7     std::cout << "Factorial de 5: " << factorial(5);
```

```
8 }
```

Funcții inline

Aceste funcții sugerează compilatorului să înlocuiască apelul funcției cu codul său intern pentru a evita suprasarcina apelurilor de funcție.

Exemplu 2.19 Exemplu de funcție inline

```
1 inline int patrat(int x) {
2     return x * x;
3 }
4
5 int main() {
6     std::cout << "Patratul lui 4 este: " << patrat(4);
7 }
```

Funcții supraincarcate

În C++, mai multe funcții pot avea același nume, atâta timp cât parametrii lor sunt diferiți.

Exemplu 2.20 Exemplu de supraincarcare a funcțiilor

```
1 int suma(int a, int b) {
2     return a + b;
3 }
4
5 double suma(double a, double b) {
6     return a + b;
7 }
8
9 int main() {
10     std::cout << suma(5, 10); // Apeleaza varianta cu int
11     std::cout << suma(2.5, 3.7); // Apeleaza varianta cu double
12 }
```

2.9.3 Transmisia parametrilor către funcții

Parametrii pot fi transmiși în trei moduri:

Transmitere prin valoare

O copie a valorii este transmisă funcției, iar modificările efectuate în funcție nu afectează variabila originală.

Exemplu 2.21 Transmitere prin valoare

```
1 void dubleaza(int x) {
2     x = x * 2; // Modificarea nu afecteaza variabila originala
3 }
4
5 int main() {
6     int numar = 5;
7     dubleaza(numar);
8     // numar ramane 5
9 }
```

Transmitere prin referință

În C++, utilizarea referințelor (&) permite modificarea directă a valorii originale.

Exemplu 2.22 Transmitere prin referință

```
1 void dubleaza(int &x) {
2     x = x * 2; // Modificarea afecteaza variabila originala
3 }
4
5 int main() {
6     int numar = 5;
7     dubleaza(numar);
8     // numar devine 10
9 }
```

Transmitere prin pointer

Funcțiile pot primi parametri sub formă de pointeri, ceea ce permite modificarea directă a valorii.

Exemplu 2.23 Transmitere prin pointer

```
1 void dubleaza(int *x) {
2     *x = (*x) * 2;
3 }
4
5 int main() {
6     int numar = 5;
7     dubleaza(&numar);
8     // numar devine 10
9 }
```

2.9.4 Memoria și gestionarea funcțiilor

În timpul execuției unui program, memoria este împărțită în mai multe segmente, fiecare având un rol bine definit. Funcțiile interacționează cu aceste segmente, iar înțelegerea modului în care memoria este utilizată este esențială pentru scrierea unui cod eficient și evitarea erorilor precum scurgerile de memorie sau depășirile de stivă.

Segmentele memoriei

Memoria procesului unui program este împărțită în mai multe regiuni:

- **Segmentul de cod (Text Segment)** – conține codul mașină al programului, adică instrucțiunile funcțiilor definite.
- **Segmentul de date (Data Segment)** – stochează variabilele globale și statice. Este împărțit în:
  - **Zona de date inițializate** – pentru variabile globale/statice inițializate.
  - **Zona de date neinițializate (BSS - Block Started by Symbol)** – pentru variabile globale/statice neinițializate.
- **Heap-ul** – utilizat pentru alocarea dinamică a memoriei (de exemplu, prin `malloc()` în C sau `new` în C++).
- **Stack-ul** – utilizat pentru stocarea apelurilor de funcții, parametrilor, variabilelor locale și adreselor de retur.

Stiva apelurilor de funcții

Atunci când o funcție este apelată, se creează un **cadru de stivă (stack frame)** care conține:

- Adresa de retur – indică unde trebuie să continue execuția după terminarea funcției.
- Parametrii funcției – valorile transmise funcției.
- Variabilele locale – variabilele definite în interiorul funcției.

La finalul execuției, cadrul de stivă este eliminat automat, iar controlul revine la apelant.

Exemplu 2.24 Exemplu de utilizare a memoriei în funcții

```
1 void functie(int x) {
2     int y = x * 2; // Variabila locala (stocata in stiva)
3     std::cout << "y = " << y << std::endl;
4 }
5
6 int main() {
7     int a = 10; // Variabila locala in stiva main
8     functie(a); // Creare si distrugere automata a cadrului de
        ↪ stiva
9 }
```

Probleme legate de memorie

Depășirea stivei (Stack Overflow:) dacă un program apelează funcții recursiv fără o condiție de oprire, stiva se poate umple și duce la o eroare de tip **stack overflow**.

Exemplu 2.25 Exemplu de depășire a stivei

```
1 void recursiv() {
2     recursiv(); // Apel recursiv infinit
3 }
4
5 int main() {
6     recursiv(); // Va duce la stack overflow
7 }
```

Scurgeri de memorie (Memory Leaks): dacă alocăm memorie pe heap și nu o eliberăm, memoria poate fi irosită și duce la un consum excesiv.

Exemplu 2.26 Exemplu de scurgere de memorie

```
1 void alocaMemorie() {
2     int* ptr = new int[100]; // Alocare pe heap
3     // Fara delete[], memoria nu este eliberata
4 }
5
6 int main() {
7     alocaMemorie();
8     // Memoria ramane alocata chiar si dupa terminarea functiei
9 }
```

Pentru a evita astfel de probleme, este esențial să folosim `delete` sau `delete[]` pentru a elibera memoria alocată pe heap.

Exemplu 2.27 Exemplu corect cu eliberare de memorie

```
1 void alocaMemorie() {
2     int* ptr = new int[100];
3     delete[] ptr; // Eliberare corecta a memoriei
4 }
```

Optimizarea utilizarii memoriei

- folosirea referințelor în loc de copiere pentru a evita alocări suplimentare.
- evitarea alocărilor frecvente pe heap în bucle.
- utilizarea `smart pointers` în C++ pentru gestionarea automată a memoriei.

Gestionarea corectă a memoriei este esențială în programele scrise în C și C++. Înțelegerea modului în care funcțiile utilizează stiva și heap-ul poate preveni erori critice precum depășirea stivei și scurgerile de memorie. Funcțiile sunt elemente fundamentale ale limbajelor C și C++, oferind modularitate, reutilizare și eficiență. Înțelegerea profundă a mecanismelor lor de memorie și a modului de transmitere a parametrilor ajută la scrierea unui cod optim și ușor de întreținut.

3. Fundamentele Python

Sintaxa de bază
Tipuri de date și variabile în Python
Operatorii în Python
Instrucțiuni de control în Python
Structuri de date în Python
Funcții în Python
Module și pachete Python
Utilizarea fișierelor în Python
Programarea orientata pe obiecte (POO) în Python
Înțelegerea excepțiilor și depanarea codului în Python
Bune practici pentru scris cod în Python

Python este un limbaj de programare robust și puternic, renumit pentru simplitatea, claritatea și ecosistemul său vast de biblioteci și cadre de lucru. Conceptul a aparut la sfârșitul anilor 1980, inițiatorul lui fiind Guido van Rossum [17]. Python a crescut de atunci într-unul dintre cele mai populare limbaje printre dezvoltatori, educatori și cercetători.

Ceea ce distinge Python-ul este sintaxa sa elegantă, care pune accent pe claritate și simplitate, făcându-l o alegere ideală pentru începători și programatori experimentați deopotrivă. Filosofia sa de design prioritizează claritatea și practicitatea, permițând dezvoltatorilor să exprime idei complexe în mai puține linii de cod.

Versatilitatea Python-ului este dată de gama sa largă de aplicații, de la dezvoltarea web și analiza datelor la inteligența artificială și calculul științific. Biblioteca sa standard extinsă oferă module și funcții încorporate pentru o gamă largă de sarcini, în timp ce comunitatea activă open-source contribuie în mod continuu cu noi biblioteci și instrumente pentru a extinde și mai mult capacitățile Python-ului.

Unul dintre punctele forte ale Python-ului constă în flexibilitatea și ușurința de a-l învăța. Sintaxa sa simplă și tipizarea dinamică permit dezvoltatorilor să prototipeze rapid idei și să itereze soluții, favorizând un ciclu rapid de dezvoltare. În plus, compatibilitatea sa cu alte platforme asigură portabilitate pe alte sisteme de operare .

Fie că sunteți un programator începător, care își începe călătoria în programare, sau un dezvoltator experimentat care caută un instrument fiabil pentru construirea de sisteme

complexe, Python-ul oferă un mediu primitor și funcții robuste pentru a satisface nevoile dumneavoastră.

3.1 Sintaxă de bază

Înțelegerea sintaxei unui limbaj de programare este crucială deoarece formează baza pe care sunt construite toate construcțiile de programare.

3.1.1 Python ca limbaj interpretat de nivel înalt

Python-ul este clasificat ca un limbaj de programare de nivel înalt, ceea ce înseamnă că abstractizează multe detalii de nivel scăzut, cum ar fi gestionarea memoriei și interacțiunea cu hardware-ul, permițând dezvoltatorilor să se concentreze mai mult pe rezolvarea problemelor decât pe complicatiile arhitecturii computerului. Această natură de nivel înalt face ca codul Python să fie ușor de citit, scris și întreținut.

Python este un limbaj de programare interpretat, ceea ce înseamnă că, codul sursă Python nu este tradus în cod de mașină înainte de executare, cum se întâmplă în cazul limbajelor compilate. În schimb, codul Python este executat de un interpretor Python linie cu linie în timpul rulării programului. Acest proces de interpretare permite dezvoltatorilor să scrie și să execute cod rapid, fără a fi nevoie să facă un pas intermediar de compilare. De asemenea, face ca dezvoltarea să fie mai interactivă, permitând programatorilor să testeze și să itereze rapid diferite părți ale codului fără a compila din nou întregul program. Python-ul este un limbaj interpretat portabil, ceea ce înseamnă că același cod Python poate rula pe diferite platforme cu un interpretor Python corespunzător instalat, fără a fi nevoie de modificări suplimentare. Acest aspect face ca Python-ul să fie o alegere populară pentru dezvoltarea rapidă și pentru prototipare, precum și pentru proiecte care necesită portabilitate și interoperabilitate.

Combinația dintre a fi de nivel înalt și interpretat oferă Python-ului mai multe avantaje, cum ar fi:

- **Ușurința învățării:** Sintaxa simplă și ușor de citit a Python-ului face ca acesta să fie ușor de învățat și de înțeles pentru începători, permițându-le să se concentreze pe învățarea conceptelor de programare în loc să lupte cu sintaxa complexă.
- **Dezvoltare rapidă:** Sintaxa concisă și tipizarea dinamică a Python-ului permit dezvoltatorilor să scrie cod rapid, să itereze rapid soluții și să aducă idei la viață mai repede în comparație cu limbajele de nivel inferior.
- **Independența de platformă:** Codul Python poate rula pe orice platformă care are instalat un interpretor Python, fie că este vorba de Windows, macOS, Linux sau chiar platforme mobile precum Android și iOS.
- **Interactivitate:** Interpretorul Python permite dezvoltarea interactivă, unde codul poate fi executat linie cu linie, făcându-l ideal pentru programarea exploratorie și depanare.

3.1.2 Indentarea și importanța spațiilor albe în Python

În Python, indentarea și spațiile albe joacă un rol crucial în definirea structurii și a domeniului de aplicare al codului. Spre deosebire de multe alte limbaje de programare care folosesc acolade sau cuvinte cheie pentru a denota blocurile de cod, Python-ul folosește indentarea pentru a indica începutul și sfârșitul blocurilor. Indentarea se referă la spațiile sau taburile plasate la începutul liniilor de cod pentru a le organiza în blocuri. Aceste blocuri includ în mod tipic instrucțiuni în cadrul structurilor de control, cum ar fi buclele, instrucțiunile condiționale și definițiile de funcții.

De exemplu, să luăm următorul fragment de cod Python:

```
1 if x > 5:
2     print("x este mai mare decat 5")
3 else:
4     print("x nu este mai mare decat 5")
```

În acest exemplu, liniile de cod din blocurile `if` și `else` sunt indentate cu patru spații. Această indentare indică faptul că aceste linii aparțin blocurilor respective. O indentare incorectă va duce la o eroare de sintaxă.

În Python, spațiile albe, inclusiv spațiile și taburile, sunt semnificative și pot afecta comportamentul codului. În mod specific, spațiile albe de la sfârșitul liniilor sunt ignorate, dar spațiile albe de la începutul liniilor sunt semnificative pentru indentare.

De exemplu, să luăm următorul fragment de cod Python:

```
1 x = 5
2 y = 10
```

Liniile `x = 5` și `y = 10` nu au spații albe la început, indicând că ele se află la același nivel de indentare și sunt executate secvențial.

Cu toate acestea, să luăm următorul fragment de cod cu o indentare incorectă:

```
1 x = 5
2   y = 10
```

În acest caz, linia `y = 10` este indentată cu un spațiu suplimentar în comparație cu `x = 5`, rezultând într-o eroare de tipul `IndentationError` deoarece Python-ul se așteaptă la o indentare consecventă în cadrul aceluiași bloc.

O indentare consecventă și corectă este esențială pentru scrierea unui cod Python curat și ușor de întreținut. Este recomandat să folosiți patru spații pentru fiecare nivel de indentare, conform ghidului de stil Python (PEP 8). De asemenea, evitați amestecarea taburilor și spațiilor pentru indentare în același fișier pentru a preveni confuzia și posibilele erori.

Înțelegerea importanței indentării și spațiilor albe în Python este fundamentală pentru scrierea unui cod curat și bine structurat, care este ușor de înțeles și întreținut.

3.1.3 Comentarii în Python

Comentariile sunt texte explicative adăugate în codul sursă pentru a explica anumite porțiuni de cod sau pentru a face acesta mai ușor de înțeles pentru alte persoane care citesc codul. În Python, există două forme de comentarii:

Comentariile pe o singură linie încep cu caracterul # și pot fi plasate oriunde pe acea linie, iar orice text de după # este ignorat de interpretorul Python.

```
1 Acesta este un comentariu pe o singura linie
2 x = 5 # Aici este un alt comentariu pe aceeasi linie
```

Pentru a comenta mai multe linii în Python, puteți folosi # pe fiecare linie, fiecare linie va fi un comentariu pe sine:

```
1 # Acesta este un comentariu
2 # pe mai multe linii
```

Alternativ, puteți folosi șiruri de documentare pentru comentarii mai lungi sau pentru a documenta funcții, clase sau module:

```
1  def add(a, b):
2      """
3      Aceasta functie aduna doua numere si returneaza rezultatul.
4
5      Args:
6          a : Primul numar.
7          b : Al doilea numar.
8
9      Returns:
10         r: Suma celor doua numere.
11     """
12     return a + b
```

În acest exemplu, sirul de documentare dintre ghilimele triple este un docstring care documenteaza functia add().

3.2 Tipuri de date și variabile în Python

Tipuri de date

Python suportă diverse tipuri de date încorporate pentru a reprezenta diferite tipuri de valori. Iată câteva exemple de tipuri de date comune în Python:

- **numere întregi** (`int`): numere întregi fără parte fracționară, precum 5, -3 sau 1000.
- **numere zecimale** (`float`): numere cu o parte fracționară, reprezentate cu un punct zecimal, precum 3.14, -0.5 sau 2.0.
- **șiruri de caractere** (`str`): secvențe de caractere închise între ghilimele simple, duble sau triple, precum "salut", 'Python' sau """șir pe mai multe linii""".

- **nooleeni** (`bool`): reprezintă valorile de adevăr `True` și `False`, folosite pentru operații logice și comparații.
- **liste** (`list`): colecții ordonate de elemente, care pot fi de diferite tipuri de date și sunt închise între paranteze pătrate, precum `[1, 2, 3]` sau `["mere", "banane", "cireșe"]`.
- **tupluri** (`tuple`): asemănătoare cu liste, dar imutabile, ceea ce înseamnă că elementele lor nu pot fi schimbate după creare și sunt închise între paranteze rotunde, precum `(1, 2, 3)` sau `("a", "b", "c")`.
- **dicționare** (`dict`): colecții de perechi cheie-valoare, unde fiecare cheie este asociată cu o valoare și sunt închise între acolade, precum `{"nume": "John", "vârstă": 30}`.

In tabelul 3.1 sunt centralizate cele mai utilizate tipuri de date în Python.

Tabela 3.1 Tipuri de date in Python

Exemplu	Tip de Date
`x = 5`	`int`
`x = 3.14`	`float`
`x = "Hello world"`	`str`
`x = 3j`	`complex`
`x= True`	`bool`
`x = [1, 2, 3]`	`list`
`x = ("rosu", "verde", "albastru")`	`tuplu`
`x = range(5)`	`range`
`x = {"nume": "Ana", "varsta": 20}`	`dict`

Variabile în Python

În Python, o variabilă este un nume care se referă la o valoare stocată în memorie. Spre deosebire de alte limbaje de programare, Python nu necesită declararea explicită a tipului de date al unei variabile. În schimb, tipul unei variabile este determinat dinamic în funcție de valoarea atribuită acesteia. Programatorul nu trebuie sa faca nimic pentru asignarea tipului datei. De exemplu:

```
1 # Atribuirea valorilor variabilelor
2 x = 5
3 nume = "John"
4 este_student = True
```

În acest exemplu, `x`, `nume` și `este_student` sunt variabile care stochează diferite tipuri de date - un număr întreg, un șir de caractere și un boolean, respectiv.

3.3 Operatorii în Python

Operatorii sunt simboluri sau cuvinte cheie care efectuează operații specifice pe unul sau mai mulți operanzi (e.g. variabile). Principalele tipuri de operatori disponibili în Python sunt: operatorii aritmetici, de comparație și logici.

3.3.1 Operatori aritmetici

Operatorii aritmetici sunt folosiți pentru a efectua operații matematice precum: adunare, scădere, înmulțire, împărțire, exponențiere și modulo.

- **adunare** (`+`): adaugă doi operanzi. Exemplu: `3 + 5` rezultă în `8`.
- **scădere** (`-`): scade al doilea operand din primul. Exemplu: `10 - 3` rezultă în `7`.
- **înmulțire** (`*`): înmulțește doi operanzi. Exemplu: `4 * 6` rezultă în `24`.
- **împărțire** (`/`): împarte primul operand la cel de-al doilea. Exemplu: `15 / 3` rezultă în `5.0`.
- **diviziune întreagă** (`//`): împarte primul operand la cel de-al doilea și returnează câtul rotunjit la cea mai apropiată valoare întreagă. Exemplu: `15 // 4` rezultă în `3`.
- **modulo** (`%`): returnează restul împărțirii primului operand la cel de-al doilea. Exemplu: `15 % 4` rezultă în `3`.
- **exponențiere** (`**`): ridică primul operand la puterea celui de-al doilea. Exemplu: `2 ** 3` rezultă în `8`.

3.3.2 Operatori de comparație

Operatorii de comparație sunt folosiți pentru a compara două valori și a determina relația dintre ele. Ei returnează o valoare booleană (Adevărat sau Fals) în funcție de comparație.

- **egal** (`==`): returnează adevărat dacă operanzii sunt egali. Exemplu: `3 == 3` rezultă în `True`.
- **diferit** (`!=`): returnează adevărat dacă operanzii nu sunt egali. Exemplu: `3 != 2` rezultă în `True`.
- **mai mare** (`>`) / **mai mic** (`<`): returnează adevărat dacă primul operand este mai mare (sau mai mic) decât al doilea. Exemplu: `5 > 3` rezultă în `True`.
- **mai mare sau egal** (`>=`) / **mai mic sau egal** (`<=`): returnează adevărat dacă primul operand este mai mare sau egal (sau mai mic sau egal) decât al doilea. Exemplu: `4 <= 4` rezultă în `True`.

3.3.3 Operatori logici

Operatorii logici sunt folosiți pentru a combina declarații condiționale și pentru a returna un rezultat boolean în funcție de operația logică efectuată.

- **ȘI logic** (`and`): returnează adevărat dacă ambii operanzi sunt adevărați. Exemplu: `True and False` rezultă în `False`.
- **SAU logic** (`or`): returnează adevărat dacă cel puțin unul dintre operanzi este adevărat. Exemplu: `True or False` rezultă în `True`.

- **NU logic** (`not`): returnează adevărat dacă operandul este Fals și Fals dacă operandul este adevărat. Exemplu: `not True` rezultă în `False`.

3.4 Instrucțiuni de control în Python

Instrucțiuni de control în Python permit programului să ia decizii și să execute anumite acțiuni în funcție de condiții sau să itereze prin secțiuni de cod în mod repetat. În continuare vom explora o serie de instrucțiuni de control.

3.4.1 Instrucțiuni condiționale (`if`, `elif`, `else`)

Instrucțiunile condiționale permit programului să execute diferite acțiuni în funcție de evaluarea unei condiții. Structura de bază a unei instrucțiuni condiționale este următoarea:

```
if conditie:
    # Blocul de cod executat daca, conditia este adevarata
elif alta_conditie:
    # Blocul de cod executat daca conditia anterioara este falsa
        ↪ si aceasta conditie este adevarata
else:
    # Blocul de cod executat daca nicio conditie anterioara nu
        ↪ este adevarata
```

Un examplu concret pentru utilizarea instrucțiunilor conditionale în Python este urmatorul:

```
x = 10
if x > 0:
    print("x este pozitiv")
elif x == 0:
    print("x este zero")
else:
    print("x este negativ")
```

3.4.2 Instrucțiuni repetitive (`for`, `while`)

Construcțiile de tip bucle permit programului să itereze prin secțiuni de cod în mod repetat. În Python, avem două tipuri principale de bucle: buclele `for` și buclele `while`.

Bucle `for`:

```
for element in secventa:
    # Blocul de cod care se repeta pentru fiecare element din
        ↪ secventa
```

Un exemplu pentru utilizarea buclei `for` în Python este urmatorul:

```
fructe = ["mere", "banane", "portocale"]
for fruct in fructe:
    print(fruct)
```

Bucle `while`:

```
while conditie:
    # Blocul de cod care se repeta atat timp cat conditia este
      ↪ adevarata
```

Un exemplu pentru utilizarea buclei `while` în Python este urmatorul:

```
i = 0
while i < 5:
    print(i)
    i += 1
```

3.4.3 Instrucțiuni `break` și `continue`

Instrucțiunea `break` este folosită pentru a ieși dintr-o buclă înainte de a fi completată, iar instrucțiunea `continue` este folosită pentru a trece la următoarea iterație a buclei, ignorând restul codului din blocul de buclă curent.

Un exemplu pentru utilizarea instrucțiunii `break` în Python este urmatorul:

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

Această buclă va printa numerele de la 0 la 4, iar apoi va ieși din buclă când `i` devine 5.

Un exemplu pentru utilizarea instrucțiunii `continue` în Python este urmatorul:

```
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
```

Această buclă va printa doar numerele impare de la 1 la 9, trecând peste numerele pare.

Vom explora principalele structuri de date disponibile în Python și modul în care acestea pot fi create, accesate și utilizate pentru a organiza și manipula datele.

3.5 Structuri de date în Python

În Python, există mai multe structuri de date fundamentale care sunt utilizate pentru a organiza și manipula datele în programele Python. Cele mai importante structuri de date sunt:

- **liste (eng. lists)**: liste ordonate de elemente, care pot fi modificate. Acestea sunt definite folosind paranteze pătrate `[]` și pot conține elemente de tip diferit. Listele sunt flexibile și pot fi modificate în mod direct.
- **tupluri (eng. tuples)**: tupluri ordonate și imutabile de elemente. Sunt definite folosind paranteze rotunde `()` și, spre deosebire de liste, nu pot fi modificate după ce sunt create.
- **dicționare (eng. dictionaries)**: dicționare neordonate de perechi cheie-valoare, în care fiecare cheie este asociată cu o valoare. Dicționarele sunt definite folosind parantezele răsucite `{}` și sunt utile pentru a stoca și accesa date în funcție de o anumită cheie.
- **seturi (eng. sets)**: seturi neordonate de elemente unice. Seturile sunt definite folosind parantezele de tip acolade `{}` și sunt utile pentru a elimina duplicatele dintr-o colecție de date și pentru a efectua operații de set, cum ar fi reuniunea, intersecția și diferența.

Aceste structuri de date sunt fundamentale în programarea Python și sunt utilizate în mod frecvent pentru a organiza și manipula datele într-un mod eficient și flexibil.

3.5.1 Liste

Listele sunt colecții ordonate de elemente, care pot fi de tipuri diferite și sunt modificate (mutable). Acestea sunt definite folosind paranteze pătrate `[]` și pot fi create astfel:

```
1 lista = [1, 2, 3, 4, 5]
```

Listele pot fi create folosind lista de elemente între paranteze pătrate. Elementele unei liste pot fi accesate folosind indexarea și segmentarea, după cum urmează:

```
1 print(lista[0])  # Afiseaza primul element din lista
2 print(lista[-1])  # Afiseaza ultimul element din lista
3 print(lista[2:4])  # Afiseaza elementele de la indexul 2 la 3
```

Python oferă o serie de metode utile pentru a lucra cu liste, cum ar fi `append()`, `insert()`, `pop()`, `remove()`, `sort()`, după cum urmeaza:

```
1 lista.append(6)  # Adauga elementul 6 la sfarsitul listei
2 lista.insert(2, 10)  # Insereaza elementul 10 la indexul 2
3 lista.pop()  # Sterge si returneaza ultimul element din lista
4 lista.remove(3)  # Sterge prima aparitie a elementului 3 din lista
5 lista.sort()  # Ordoneaza lista in ordine crescatoare
```

3.5.2 Tuple

Tuplele sunt colecții ordonate și imutabile de elemente, definite folosind paranteze rotunde `()`. Tuplurile pot fi create folosind lista de elemente între paranteze rotunde, după cum urmeaza:

```
tuplu = (1, 2, 3, 'patru', 'cinci')
```

Elementele unui tuplu pot fi accesate folosind indexarea, după cum urmează:

```
print(tuplu[0])  # Afiseaza primul element din tuplu
print(tuplu[-1])  # Afiseaza ultimul element din tuplu
```

3.5.3 Dicționare

Dicționarele sunt colecții neordonate de perechi cheie-valoare, definite folosind parantezele răsucite `{}`. Dicționarele pot fi create specificând perechile cheie-valoare între paranteze acolade, după cum urmează:

```
student = {'nume': 'Ana', 'varsta': 20, 'nota': 9.5}
```

Elementele dintr-un dicționar sunt organizate în perechi cheie-valoare, unde cheile sunt unice și sunt folosite pentru a accesa valorile corespunzătoare, după cum urmează:

```
print(student['nume'])  # Afiseaza valoarea asociata cheii 'nume'
```

3.5.4 Seturi

Seturile sunt colecții neordonate de elemente unice, definite folosind paranteze de tip acolade `{}`. Seturile pot fi create specificând elementele unice între parantezele de tip acolade, după cum urmează:

```
setul_meu = {1, 2, 3, 4, 5}
```

Elementele dintr-un set sunt unice și duplicatele sunt eliminate automat, după cum urmează:

```
setul_meu.add(6)  # Adauga elementul 6 in set
setul_meu.remove(3)  # Sterge elementul 3 din set
```

3.6 Funcții în Python

Funcțiile reprezintă un concept fundamental în programare, inclusiv în limbajul de programare Python. Ele servesc la organizarea și structurarea codului în blocuri logice și reutilizabile, aducând o serie de beneficii și îndeplinind diverse scopuri în dezvoltarea de software:

- **Reutilizabilitatea codului**: Funcțiile permit gruparea logică a unor secțiuni de cod care efectuează o anumită acțiune și pot fi apelate de oriunde în program. Astfel, codul poate fi scris o singură dată și utilizat în mai multe locuri.
- **Modularitatea**: Funcțiile permit împărțirea unui program mare în părți mai mici și mai ușor de gestionat. Acest lucru face dezvoltarea, testarea și întreținerea codului mai eficiente.
- **Abstracția**: Funcțiile permit definirea unor nume semnificative pentru anumite acțiuni sau operații, ceea ce face codul mai ușor de înțeles și de gestionat.

- **Încapsularea**: Funcțiile permit izolarea unor anumite logici sau operații într-un context separat, reducând complexitatea și creând un nivel de abstractizare între diferitele părți ale codului.

În Python, funcțiile sunt utilizate pentru a efectua o gamă largă de activități, cum ar fi procesarea datelor, manipularea șirurilor de caractere, realizarea de calcule matematice, comunicarea cu bazele de date, crearea de interfețe grafice și multe altele. Python oferă și un set bogat de funcții predefinite, precum și posibilitatea de a defini funcții personalizate, ceea ce face limbajul versatil și puternic pentru dezvoltarea de software. Prototipul unei funcții în Python este următorul:

```
1  def aduna(param1, param2):
2      """
3      Descriere: Aceasta este o functie de exemplu care primeste doi
           ↪ parametri si returneaza rezultatul sumei lor.
4
5      Args:
6      param1 (tip): Descriere parametru 1.
7      param2 (tip): Descriere parametru 2.
8
9      Returns:
10     tip: Descriere a ceea ce returneaza functia.
11     """
12     # Corpul functiei
13     result = param1 + param2
14     return result
```

3.6.1 Functia predefinita main()

Funcția main() este o convenție în programarea în limbajul C și este folosită pentru a marca punctul de intrare în program. Această convenție a fost preluată și în alte limbaje de programare, inclusiv în Python.

În Python, if __name__ == "__main__": este folosit pentru a verifica dacă scriptul Python este rulat direct, nu importat ca modul în alt script. Atunci când un script Python este rulat direct, interpreterul Python setează __name__ la valoarea "__main__". Astfel, blocul de cod sub condiția if __name__ == "__main__": este executat numai atunci când scriptul este rulat direct.

Este considerată o practică bună să plasezi logica principală a programului în funcția main(), deoarece acest lucru face codul mai organizat și mai ușor de înțeles. Funcția main() poate conține apelurile altor funcții și alte instrucțiuni necesare pentru a executa programul. Aceasta separă logica principală de alte definiri de funcții sau variabile globale, ceea ce face codul mai modular și mai ușor de întreținut.

Deși nu este strict necesară, utilizarea funcției main() este considerată o practică bună de

dezvoltare deoarece promovează un cod mai curat și mai organizat. De asemenea, permite ca scripturile Python să fie utilizate ca module în alte proiecte, fără a rula automat codul din main() atunci când sunt importate.

Exemplu 3.1 Utilizarea a functiei `main` pentru adunarea a 2 valori.

```
1  def aduna(param1, param2):
2      """
3      Descriere: Aceasta este o functie de exemplu care primeste doi
           ↪ parametri si returneaza rezultatul lor.
4
5      Args:
6      param1 (tip): Descriere parametru 1.
7      param2 (tip): Descriere parametru 2.
8
9      Returns:
10     tip: Descriere a ceea ce returneaza functia.
11     """
12     # Corpul functiei
13     result = param1 + param2
14     return result
15
16 def main():
17     # Apelul functiei si afisarea rezultatului
18     print("Apelul functiei:")
19     param1 = 5
20     param2 = 10
21     result = aduna(param1, param2)
22     print("Rezultatul functiei:", result)
23
24 if __name__ == "__main__":
25     main()
```

La rularea exemplului 3.1, efectul afisat în consola este urmatorul:

```
Apelul functiei:
Rezultatul functiei: 15
```

3.6.2 Domeniul variabilelor (global vs. local)

Variabilele definite în interiorul unei funcții sunt locale funcției respective și sunt valabile doar în interiorul acelei funcții. În schimb, variabilele definite în afara oricărei funcții sunt globale și pot fi accesate de oriunde în codul programului.

Exemplu:

```
x = 10  # Variabila globala

def afisare():
    y = 20  # Variabila locala
    print("Valoarea lui x este:", x)
    print("Valoarea lui y este:", y)

afisare()
```

În acest exemplu, variabila `x` este globală și poate fi accesată de funcția `afișare()`, în timp ce variabila `y` este locală și este valabilă doar în interiorul funcției `afișare()`.

3.7 Module și pachete Python

Modulele și pachetele sunt componente esențiale în dezvoltarea software-ului Python, permițând organizarea și structurarea codului în unități logice și reutilizabile. În continuare vom prezenta importarea și utilizarea modulelor, crearea și organizarea pachetelor și utilizarea bibliotecilor terțe prin pip (unde pip reprezinta sistemul de gestionare a pachetelor în Python, utilizat pentru instalarea, actualizarea și dezinstalarea bibliotecilor externe. Acesta permite accesul la depozitul central `PyPI` (Python Package Index). Comenzi uzuale includ `pip install nume_pachet`, `pip uninstall nume_pachet` și `pip list`, pentru a vizualiza pachetele instalate).

3.7.1 Importarea modulelor

În Python, modulele sunt fișiere .py care conțin definiții de funcții, clase și alte obiecte. Pentru a folosi codul dintr-un modul în alt fișier Python, putem importa acel modul folosind instrucțiunea `import`.

```
# Exemplu de importare a unui modul
import math

# Utilizarea unei functii din modulul math
print(math.sqrt(25))  # Rezultat: 5.0
```

3.7.2 Crearea și organizarea pachetelor

Pachetele sunt directoare care conțin module Python și un fișier special __init__.py care indică faptul că directorul respectiv ar trebui tratat ca un pachet Python.

```
# Structura unui pachet
mypackage/
    __init__.py
    module1.py
```

```
5       module2.py
```

Pentru a accesa modulele dintr-un pachet, putem utiliza notația punctată:

```
1 # Exemplu de utilizare a modulelor dintr-un pachet
2 import mypackage.module1
3 from mypackage import module2
4
5 mypackage.module1.function1()
6 module2.function2()
```

3.7.3 Utilizarea bibliotecilor prin `pip`

Python oferă o bogată colecție de biblioteci standard, dar, uneori, este necesar să utilizăm biblioteci terțe pentru a adăuga funcționalități suplimentare la aplicațiile noastre. Pentru a instala astfel de biblioteci, putem folosi `pip`, care este un sistem de gestionare a pachetelor Python. comanda `pip` se rulează în consola sistemului de operare, după cum urmează:

```
1 # Exemplu de instalare a unei biblioteci terte folosind pip
2 pip install numPy
```

După ce o bibliotecă terță este instalată, o putem importa și folosi în codul nostru Python:

```
1 # Exemplu de importare si utilizare a unei biblioteci terte
2 import numpy as np
3
4 array = np.array([1, 2, 3, 4, 5])
5 print(array)  # Rezultat: [1 2 3 4 5]
```

3.8 Utilizarea fișierelor în Python

Manipularea fișierelor în Python este un aspect esențial al programării, permițând citirea și scrierea datelor din fișiere externe. În acest capitol, vom explora deschiderea, citirea și scrierea în fișiere, precum și gestionarea excepțiilor cu blocurile try-except.

3.8.1 Deschiderea, citirea și scrierea în fișiere

Pentru a deschide un fișier în Python, putem utiliza funcția `open()`, specificând calea către fișier și modul de deschidere (de exemplu, `'r'` pentru citire, `'w'` pentru scriere, `'a'` pentru adăugare etc.).

```
1 # Exemplu de deschidere si citire dintr-un fisier
2 with open('example.txt', 'r') as file:
3     content = file.read()
4     print(content)
```

Pentru scrierea într-un fișier, deschiderea fișierului cu modul `'w'` sau `'a'` (pentru adăugare) permite scrierea de conținut în el.

```
# Exemplu de deschidere si scriere intr-un fisier
with open('example.txt', 'w') as file:
    file.write("Aceasta este o linie de text.\n")
    file.write("Aceasta este o alta linie de text.\n")
```

3.9 Programarea orientată pe obiecte (POO) în Python

După cum a fost prezentat și în Capitolul 1, programarea orientată pe obiecte (POO) este o paradigma de programare care utilizează obiecte și clase pentru a organiza și structura codul. În Python, POO este un aspect esențial al limbajului, permițând dezvoltatorilor să creeze cod modular și mai ușor de întreținut. Clasele și obiectele, atributele și metodele, moștenirea și polimorfismul sunt câteva dintre conceptele de bază ale POO în Python. Aceste concepte vor fi descrise pe larg în continuarea acestei cărți.

3.9.1 Clase și obiecte

O clasă în Python este o modalitate pentru crearea obiectelor. Clasele definesc proprietățile și comportamentele pe care obiectele create din ele le vor avea. În Python, definiți o clasă folosind cuvântul cheie `class`, urmat de numele clasei și de caracterul două puncte (:). Proprietățile și comportamentele sunt definite în clasă folosind atribute și metode.

```
class Caine:
    def __init__(self, nume, varsta):
        self.nume = nume
        self.varsta = varsta

    def latra(self):
        return f"{self.nume} spune ham!"
```

Obiectele sunt instanțe ale claselor. Odată ce aveți o clasă, puteți crea obiecte din ea. Crearea unui obiect este cunoscută și sub numele de instanțierea unei clase. Fiecare obiect are propriul set de atribute și metode așa cum este definit în clasă, dar valorile acestor atribute pot fi unice pentru fiecare obiect.

```
cainele_meu = Caine("Buddy", 5)
print(cainele_meu.latra())  # Buddy spune ham!
```

3.9.2 Atribute și metode

Atributele sunt variabile care aparțin unei clase sau unei instanțe a unei clase. Ele reprezintă starea sau caracteristicile unui obiect. În exemplul clasei `Caine`, `nume` și `varsta` sunt atribute ale clasei.

Metodele sunt funcții care aparțin unei clase. Ele definesc comportamentele obiectelor create din clasă. Metodele sunt apelate pe obiecte folosind notația cu punct. În clasa noastră `Caine`, `latra` este o metodă care definește un comportament al câinelui.

3.9.3 Moștenire și polimorfism

Moștenirea permite unei clase să moștenească atribute și metode de la o altă clasă, facilitând reutilizarea codului și crearea unei ierarhii de clase. Clasa care moștenește este numită clasa copil, iar clasa de la care moștenește este clasa părinte.

```
class Labrador(Caine):
    def __init__(self, nume, varsta, culoare):
        super().__init__(nume, varsta)
        self.culoare = culoare

    def aduce(self):
        return f"{self.nume} aduce mingea!"
```

Polimorfismul permite metodelor să facă lucruri diferite bazate pe obiectul care le apelează. Acest lucru înseamnă că o metodă aparținând unei clase poate fi utilizată de obiecte ale diferitelor clase. Polimorfismul este strâns legat de moștenire, deoarece permite claselor copil să suprascrie sau să extindă funcționalitatea metodelor clasei părinte.

```
def vorbeste_caine(caine):
    print(caine.latra())

labrador = Labrador("Max", 3, "auriu")
vorbeste_caine(labrador)  # Max spune ham!
```

3.10 Înțelegerea excepțiilor și depanarea codului în Python

Când Python întâlnește o eroare în timpul execuției, acesta se oprește și afișează un mesaj de eroare cunoscut sub numele de excepție. Excepțiile reprezintă modul în care Python semnalează că ceva a mers greșit. Înțelegerea acestor mesaje de eroare, sau "traceback-uri", este crucială pentru diagnosticarea și corectarea problemelor din cod.

Un traceback oferă informații valoroase despre ce s-a întâmplat, inclusiv tipul erorii, linia de cod unde a apărut eroarea și secvența de apeluri de funcții care au condus la eroare. Aceste informații sunt esențiale pentru a localiza unde în codul tău se află problema.

3.10.1 Excepții comune în Python

Înțelegerea diferitelor tipuri de excepții este esențială pentru a scrie cod robust și pentru a gestiona erorile în mod eficient. Iată câteva dintre cele mai comune tipuri de excepții în Python:

- **SyntaxError**: apare atunci când Python întâlnește o eroare de sintaxă. De exemplu,

poate fi cauzată de lipsa unui ':' la sfârșitul unei instrucțiuni 'if'.

- **IndentationError**: se generează când există probleme cu indentarea codului. Python se bazează pe indentare pentru a defini blocurile de cod.
- **NameError**: apare când o variabilă sau o funcție care nu a fost definită este accesată. Acest lucru se poate întâmpla dacă încerci să folosești o variabilă înainte de a o defini.
- **TypeError**: este generată atunci când o operație sau funcție este aplicată unui obiect de un tip inadecvat. De exemplu, încercând să adaugi un număr la un șir de caractere.
- **ValueError**: această excepție este generată când o funcție primește un argument cu tipul corect, dar cu o valoare inadecvată.
- **IndexError**: se întâmplă atunci când încerci să accesezi un index dintr-o listă sau un tuplu care nu există.
- **KeyError**: similar cu IndexError, dar pentru dicționare. Apare atunci când încerci să accesezi o cheie care nu există în dicționar.
- **AttributeError**: este generată atunci când o referință de atribut sau atribuire eșuează. De exemplu, atunci când încerci să accesezi un atribut al unui obiect care nu există.
- **IOError**: apare atunci când o operație de intrare/ieșire eșuează, de exemplu, la încercarea de a deschide un fișier care nu există.
- **ZeroDivisionError**: este generată atunci când al doilea operand al unei operații de împărțire sau al modulo este zero.

Gestionarea acestor excepții în mod adecvat este crucială pentru asigurarea stabilității și fiabilității programelor Python. Utilizarea blocurilor `try` și `except` permite programatorilor să anticipeze și să gestioneze erorile, oferind în același timp o experiență de utilizare netulburată.

3.10.2 Gestionarea excepțiilor utilizând `try` și `except`

Pentru a gestiona excepțiile și a preveni închiderea neașteptată a programului tău, poți utiliza blocul `try` și `except`. Iată o structură de bază:

```
try:
    # Cod care ar putea genera o exceptie
except TipulExceptiei:
    # Cod care se executa daca apare exceptia
```

Prin prinderea excepțiilor, poți oferi utilizatorilor feedback util sau poți lua măsuri corective în loc să lași programul să se termine neașteptat. În continuare, este detaliat un examplu de gestionare a exceptiilor:

```
# Exemplu de gestionare a unei exceptii
try:
    num = int(input("Introduceti un numar: "))
    print("Dublul numarului introdus este:", num * 2)
except ValueError:
    print("Ati introdus o valoare nevalida pentru numar.")
```

În acest exemplu, blocul `try` încearcă să convertească inputul utilizatorului într-un număr întreg. Dacă conversia eșuează (de exemplu, dacă utilizatorul introduce un șir de caractere care nu poate fi convertit în număr), blocul `except` este executat, afișând un mesaj corespunzător.

3.10.3 Tehnici de depanare

Depanarea este procesul de identificare și rezolvare a bug-urilor (defecțiuni sau probleme care împiedică funcționarea corectă) în cadrul programelor de calculator, rețelelor sau sistemelor.

Una dintre tehnicile simple, dar eficiente, este utilizarea instrucțiunilor `print` pentru a urmări fluxul programului tău și valorile variabilelor în diferite puncte. Este o metodă directă care nu necesită unelte suplimentare.

Pentru probleme mai complexe, Python oferă mai multe unelte de depanare, cum ar fi:

- **pdb (Python Debugger)**: un depanator interactiv puternic care îți permite să parcurgi codul pas cu pas, să examinezi variabile și să evaluezi expresii.
- **Depanatoare IDE**: mediile de dezvoltare integrate (IDE-uri) precum PyCharm, Visual Studio Code și Eclipse cu PyDev oferă capacități de depanare integrate, cu interfețe prietenoase utilizatorului.

Utilizând aceste unelte, se pot seta puncte de oprire care permit: parcurgerea codului pas cu pas, inspecta valorile variabilelor și multe altele. Acest lucru permite o abordare mai eficientă și sistematică a depanării, comparativ cu utilizarea singură a instrucțiunilor print.

3.11 Bune practici pentru scris cod în Python

În Python, ca în toate limbajele de programare, scrierea unui cod care funcționează este doar primul pas. Pasul următor, la fel de important, este scrierea unui cod curat, lizibil și întreținut. Acest capitol explorează cele mai bune practici recomandate de comunitatea Python, cu un accent special pe PEP 8, ghidul de stil pentru codul Python.

3.11.1 Convenții de stil al codului

Respectarea conventiilor de stil al codului îmbunătățește lizibilitatea codului, facilitând înțelegerea acestuia de către alții (și de către tine însuți, mai târziu). De asemenea, facilitează colaborarea între mai mulți dezvoltatori care lucrează la același proiect.

Indentarea

Python folosește indentarea pentru a marca blocurile de cod. Este important ca indentarea să fie consistentă. De obicei, se folosesc 4 spații pentru fiecare nivel de indentare (nu taburi).

```
1 def calculeaza_suma(a, b):
2     suma = a + b
```

```
3       return suma
```

Nume pentru variabile și funcții

Se recomandă utilizarea stilului `snake_case` pentru numele funcțiilor și variabilelor, adică litere mici și cuvinte separate prin underscore (_).

```
1 valoare_maxima = 100
2 def calculeaza_media(valoare1, valoare2):
3     return (valoare1 + valoare2) / 2
```

Comentarii

Comentariile trebuie să fie clare și să explice secțiuni de cod care nu sunt evidente. De asemenea, se recomandă comentarii pe linii separate față de codul propriu-zis.

```
1 # Calculam media aritmetica
2 def calculeaza_media(valoare1, valoare2):
3     return (valoare1 + valoare2) / 2
```

Comentariile care explică funcționalitatea unei funcții ar trebui să fie plasate înainte de funcția respectivă:

```
1 def calculeaza_media(valoare1, valoare2):
2     """
3     Functia calculeaza media a doua valori.
4     :param valoare1: prima valoare
5     :param valoare2: a doua valoare
6     :return: media celor doua valori
7     """
8     return (valoare1 + valoare2) / 2
```

Maxim de 79 de caractere pe linie

Este recomandat ca fiecare linie de cod să nu depășească 79 de caractere. În cazurile în care este necesar să împărțiți o linie lungă, o puteți face astfel:

```
1 def calculeaza_media(valoare1, valoare2):
2     return (
3         valoare1 + valoare2
4     ) / 2
```

Spațierea între operatori

Se recomandă adăugarea unui spațiu înainte și după operatori (ex: `+`, `-`, `=`, `>`, `<` etc.).

```
a = 5 + 3
b = a * 2
```

Numele claselor

Pentru numele claselor se folosește stilul `CamelCase`.

```
class CalculatorClasic:
    def __init__(self):
        self.valoare = 0

    def adauga(self, numar):
        self.valoare += numar
```

Importuri

Importurile trebuie să fie pe linii separate și ar trebui să urmeze un anumit tipar: biblioteci standard, biblioteci externe și biblioteci proprii ale proiectului.

```
import os
import sys

from numpy import array
```

Folosirea ghilimelelor

În Python, poți folosi atât ghilimele simple (’), cât și ghilimele duble ("), dar este important să fii consecvent. De obicei, pentru un text simplu, se folosesc ghilimele duble:

```
mesaj = "Salut, lume!"
```

3.11.2 Scrierea unui cod curat, lizibil și ușor de întreținut

Scrierea unui cod lizibil și ușor de întreținut este crucială. Iată câteva linii directoare:

- scrieți codul ca și cum comentariile nu ar exista. Codul dvs. ar trebui să fie auto-explicativ.
- evitați structurile complexe și încâlcite. Simplitatea este cheia lizibilității.
- folosiți comentariile pentru a explica "de ce" se face ceva, nu "ce" se face. Codul în sine ar trebui să explice "ce".
- mențineți comentariile actualizate. Comentariile înșelătoare sunt mai rele decât lipsa comentariilor.
- utilizați docstrings pentru module, funcții, clase și metode pentru a descrie ce fac.
- fiți consecvenți în utilizarea convențiilor în cadrul unui proiect. Dacă contribuiți la un proiect existent, urmați convențiile care sunt deja în vigoare.

- refactorizați regulat codul pentru a identifica și aplica îmbunătățiri. Acest lucru poate însemna descompunerea funcțiilor mari, reducerea duplicării sau îmbunătățirea denumirilor pentru claritate.

Simplificarea structurilor complexe:

```
# In loc de:
def verifica(numar):
    if numar > 0 and numar < 10:
        return True
    else:
        return False

# Utilizati:
def verifica(numar):
    return 0 < numar < 10
```

Utilizarea comentariilor pentru clarificări:

```
# In loc de:
# Verifica daca numarul este pozitiv
def pozitiv(n):
    return n > 0

# Utilizati:
def numar_pozitiv(numar):
    # Returneaza True daca numarul este mai mare decat zero
    return numar > 0
```

3.11.3 Urmărirea propunerilor de îmbunătățire Python (PEPs)

PEPs sunt documente de design care oferă informații comunității Python sau descriu o nouă caracteristică pentru Python sau procesele sale. În timp ce PEP 8 se concentrează pe ghidurile de stil, alte PEP-uri propun caracteristici noi semnificative sau îmbunătățiri ale limbajului.

- **PEP 20, Zenul Python**: oferă 19 aforisme pentru scrierea unui cod Python bun, subliniind simplitatea și lizibilitatea.
- **PEP 484, Sugestii de Tip**: introduce sugestii de tip opționale, permițând cod Python tipizat static.

Exemplu pentru aplicarea PEP 484 - Sugestii de Tip:

Adăugarea sugestiilor de tip:

```
# In loc de:
def aduna(a, b):
    return a + b
```

```
4
5 # Utilizati:
6 def aduna(a: int, b: int) -> int:
7     return a + b
```

4. Clase și obiecte

Definirea claselor în C++
Definirea claselor în Python
Variabilele și metodele membru
Constructorul și destructorul

4.1 Definirea claselor în C++

Programarea orientată pe obiecte este o metodă de implementare în care programele sunt organizate ca şi colecţii de obiecte ce cooperează între ele, fiecare obiect reprezentând instanţa unei clase; fiecare clasă aparţine unei ierarhii de clase, clasele fiind unite prin relaţii de moştenire. În aceste limbaje, obiectele şi nu algoritmii sunt blocurile logice fundamentale.

O **clasă** reprezintă un tip de date personalizat care combină o structură de date cu un set de funcții asociate. Astfel, o clasă poate fi descrisă ca fiind compusă din **date** și **operații (metode)**.

În C++, concepte precum domeniul și durata de viață, variabilele locale și globale, precum și variabilele statice, automatice și dinamice se aplică și obiectelor. Deosebit față de alte limbaje de programare orientate pe obiecte, C++ permite controlul accesului atât asupra datelor membre, cât și asupra funcțiilor membre ale unei clase. Acest control se realizează prin specificatori de acces: `private`, `protected` și `public` [16].

Efectele fiecăruia dintre acești specificatori sunt următoarele:

- **public**: membrii pot fi accesați de orice funcție din domeniul clasei.
- **private**: membrii sunt accesibili doar funcțiilor membre și funcțiilor prietene ale clasei.
- **protected**: membrii sunt accesibili funcțiilor membre și funcțiilor prietene ale clasei, dar și funcțiilor membre ale claselor derivate.

O funcție membră a unei clase poate accesa toate datele membre ale oricărui obiect din acea clasă, indiferent de specificatorul de acces folosit.

În C++, clasele pot fi declarate cu ajutorul cuvintelor cheie `struct`, `union` sau `class`, iar obiectele de tipurile respective sunt denumite **obiecte struct**, **obiecte union** și **obiecte class**.

Tipurile `class` sunt utilizate mai frecvent, deoarece reflectă mai fidel conceptul de programare orientată pe obiecte (OOP) [10].

4.1.1 Tipul class

Declarația unui tip de date `class` urmează o sintaxă generală similară cu cea a tipului `struct`:

```
1 class <nume_clasa> : <lista_clase_baza> {<lista_membri>} <
    ↪ lista_variabile>
2 {
3 }
```

unde:

- **nume_clasa**: reprezintă un identificator care definește numele tipului de clasă și trebuie să fie unic în cadrul domeniului în care declarația este valabilă.
- **lista_clase_baza**: este un set de clase din care clasa declarată este derivată (dacă este cazul).
- **lista_membri**: este o secvență care conține declarațiile datelor membre și ale metodelor (funcțiilor membre) ale clasei.
- **lista_variabile**: se referă la lista variabilelor de tipul `nume_clasa`.

lista_ membri poate conţine cele două categorii de membri, date şi respectiv funcţii membre, însoţite de specificatorii de acces. Datele membre ale clasei se declară prin tip şi nume cu sintaxa:

```
specificator_de_acces:
   tip_date nume_membru;
   tip_date nume_membru;
   ...
```

Datele membre pot avea orice tip, cu excepția tipului de clasă declarat, însă se acceptă pointeri către acesta. Astfel, datele membre pot fi de tipuri predefinite, cum ar fi `char`, `int`, `float`, etc., pot fi pointeri sau tablouri de date, sau chiar tipuri definite de utilizator, cum ar fi structuri, clase, etc. Utilizarea specificatorilor `auto`, `extern` și `register` nu este permisă.

Funcţiile membre ale clasei se declară în interiorul declaraţiei clasei prin prototipul funcţiei (tip, nume, lista parametri) sau prin definiţia completă a funcţiei, folosind sintaxa:

```
specificator_de_acces:
    tip nume_funcţie(lista_parametri);  // prototip functie
    tip nume_funcţie(lista_parametri) // definiţie functie
    { // lista instructiuni
       ....
    }
    ...
```

În mod implicit, membrii unei clase au accesibilitatea `private`.

În cadrul declarației unei clase, specificatorii de acces pot apărea de mai multe ori și în orice ordine, iar toți membrii declarați după un specificator vor avea accesul controlat de acesta. De obicei, metodele asociate datelor (funcțiile membre) trebuie să fie accesibile utilizatorului și, prin urmare, se declară cu acces public, însă pot exista situații în care să fie necesare funcții private, care pot fi apelate doar de alte funcții membre ale clasei sau de funcțiile prietene ale acesteia.

În declarația clasei se pot include fie definițiile complete ale funcțiilor membre, fie doar prototipurile acestora, iar definițiile lor pot fi plasate oriunde în proiect, chiar și în alte fișiere. Funcțiile membre definite în declarația clasei sunt, implicit, de tip `inline`. Dacă definițiile funcțiilor se află în afara declarației clasei, este necesar să se precizeze numele clasei urmat de operatorul de rezoluție `::`, indicând compilatorului că funcția respectivă face parte din aceeași clasă, permițând accesul direct la membrii acesteia. În caz contrar, compilatorul va considera că se definește o funcție externă clasei respective.

Funcțiile membre ale unei clase pot fi supradefinite și pot avea parametri impliciți. Pentru a apela funcțiile membre publice și pentru a accesa datele membre cu acces public ale unui obiect, se utilizează operatorii de selecție `.` și `->`, la fel ca în cazul structurilor și uniunilor.

Pentru a exemplifica declarația unei clase, utilizarea obiectelor de tip `class` și a membrilor acestora, precum și a funcțiilor și datelor, se poate defini clasa `Punct`, care reprezintă un punct în plan prin coordonate carteziene, având doi membri de tip `int`, `x` și `y`. Clasa include și câteva funcții membre: `init()` pentru a seta valorile inițiale ale coordonatelor punctului, `getx()` și `gety()` pentru citirea valorilor acestora, `move()` pentru modificarea coordonatelor punctului și `afisare()` pentru afișarea proprietăților punctului.

Declaraţia clasei Punct, cu membrii descrişi anterior, poate fi următoarea:

Exemplu 4.1 Declaraţia clasei Punct

```
1  #include <iostream>
2  class Punct
3  {
4  private: // se specifica accesul private la membrii clasei
5      int x, y; // date membre ale clasei
6  public: // se specifica acces public la membrii clasei
7      void init(int initx=0, int inity=0) // functie de initializare
           ↪ , functie membra cu
8      {                          // parametri impliciti
9              x = initx;
10         y = inity;
11 }
12     int getx() // functie inline, returneaz valoarea membrului x
13     {
14         return x;
```

```
15      }
16      int gety() // functie inline, returneaza valoarea membrului y
17      {
18          return y;
19      }
20      void move(int dx, int dy); // functie membra cu parametri,
           ↪ definita in afara
21                                 // declaratiei clasei
22      void afisare() // functie de afisare a valorilor membrilor,
           ↪ functie inline
23      {
24          std::cout<<"\nx="<<x<<"\ty="<<y;
25      }
26  };
27
28  void Punct::move(int dx, int dy) // definirea functiei move(),
       ↪ membra a clasei Punct
29  {
30      x+=dx;
31      y+=dy;
32  }
33
34  int main()
35  {
36      Punct Punct1;
37      return 0;
38  }
```

Funcţia membră move() se defineşte în afara declaraţiei clasei, folosind sintaxa următoare:

```
1  void Punct::move(int dx, int dy) // definirea functiei move(),
       ↪ membra a clasei Punct
2  {
3      x+=dx;
4      y+=dy;
5  }
```

Obiectele de tip Punct se declară, precizând tipul şi numele lor, de exemplu:

```
1  Punct Punct1; // se declara un obiect de tip Punct, Punct1
```

Obiectul Punct1 este alocat în memorie, în funcţie de locul în care este făcută declaraţia, în segmentul de date dacă declaraţia este globală, situaţie în care membrii date sunt iniţializaţi implicit cu 0, sau pe stivă, dacă declaraţia este locală unei funcţii, situaţie în care

membrii date preiau valori reziduale.

În continuare este exemplificat modul de utilizare a membrilor obiectelor de tip Punct:

Exemplu 4.2 Modul de utilizare a membrilor obiectelor de tip Punct

```
1  int main()
2  {
3      Punct Punct1; // se declara un obiect de tip Punct, Punct1
4      int x1, y1;
5
6      std::cout<<"\n Introduceti coordonata x= ";
7      std::cin>>x1;
8      std::cout<<" Introduceti coordonata y= ";
9      std::cin>>y1;
10
11     Punct1.init(x1, y1); // initializarea obiectului Punct1 cu
           ↪ valorile x1, respectiv y1
12     std::cout<<"\n x este = "<<Punct1.getx(); // afisarea
           ↪ coordonatei x
13     std::cout<<"\n y este = "<< Punct1.gety(); // afisarea
           ↪ coordonatei y
14     Punct1.move(10, 20); // modificarea coordonatelor obiectului
           ↪ Punct1
15     std::cout<<"\n x este = "<<Punct1.getx(); // afisarea
           ↪ coordonatei x
16     std::cout<<"\n y este = "<< Punct1.gety(); // afisarea
           ↪ coordonatei y
17
18     Punct Punct2; // se declara un obiect de tip Punct, Punct2
19     Punct2.init(); // membrii x si y preiau valorile implicite ale
           ↪  parametrilor, deci x=y=0
20     Punct2.afisare(); // afisarea caracteristicilor obiectului
           ↪ Punct2
21     Punct *p_Punct3; // se declara un pointer la Punct care poate
           ↪  prelua adresa unui obiect de tip Punct
22     p_Punct3 = &Punct2;
23     p_Punct3 -> move ( 5, 12); // apelul functiilor membre se face
           ↪  folosind  operatorul de selectie "->"
24     p_Punct3 -> afisare();
25
26     return 1;
27 }
```

La rularea exemplului 4.2, efectul afisat în consola este urmatorul:

```
 Introduceti coordonata x= 5
 Introduceti coordonata y= 6

 x este = 5
 y este = 6
 x este = 15
 y este = 26
x=0     y=0
x=5     y=12
```

Obiectele de tip `Punct` (de exemplu, `Punct1`, `Punct2`) sunt structuri de date formate din doi membri de tip `int`, `x` și `y`, care pot fi gestionate prin funcțiile membre asociate, ce permit atribuirea valorilor, afișarea și modificarea acestora (`init()`, `afisare()`, `getx()`, `gety()`, `move()`).

Accesul direct la membrii `x` și `y` nu este permis din afara clasei, aceștia putând fi manipulați doar prin intermediul funcțiilor membre, datorită specificatorului `private` ce restricționează accesul la aceștia.

```
1 Punct1.x = 15; // eroare, membrul Punct1.x este privat
2
3 Punct2.y = Punct1.x; // eroare Punct1.x, Punct2.y sunt membri
    ↪ privati
```

Pentru fiecare obiect al clasei se alocă spaţiul de memorie necesar datelor membre.

Operatorul de atribuire (=) poate fi folosit pentru obiecte de acelaşi tip, determinând copierea datelor membru cu membru.

```
1 Punct2=Punct1; //membrul x al obiectului Punct2 primeste valoarea
2 // membrului x al obiectului Punct1 membrul y al
3 // obiectului Punct2 primeste valoarea membrului y al
4 // obiectului Punct1
5 Punct2.afisare();
```

Operatorii `new` și `delete` pot fi utilizați pentru crearea și distrugerea obiectelor dinamice de tip `class`.

```
1 Punct * p =new Punct;
2 p ->init(5,7);
3 p ->move(2, 2);
4 p ->afisare();
5 delete p_Punct4;
```

4.1.2 Constructorii

Un **constructor** este o funcție membră specială care este apelată automat în momentul creării unui obiect al clasei. Constructorii au același nume ca și clasa și nu au un tip de returnare (nici măcar `void`).

Există mai multe tipuri de constructori în C++:

- **Constructor implicit**: nu primește parametri sau toți parametrii au valori implicite.
- **Constructor parametrizat**: permite inițializarea obiectului cu valori specifice.
- **Constructor de copiere**: creează un nou obiect ca o copie a altui obiect al aceleiași clase.
- **Constructor de mutare**: permite mutarea resurselor de la un obiect temporar către un alt obiect.

Exemplu de constructori:

Exemplu 4.3 Modul de utilizare la constructorilor in clasa Punct

```
#include <iostream>
class Punct {
    private:
        int x, y;
    public:
        // Constructor implicit
        Punct() : x(0), y(0) {}

        // Constructor parametrizat
        Punct(int initX, int initY) : x(initX), y(initY) {}

        // Constructor de copiere
        Punct(const Punct &p) : x(p.x), y(p.y) {}

        void afisare() {
            std::cout << "x=" << x << " y=" << y << std::endl;
        }
};

int main() {
Punct p1; // Apelul constructorului implicit
Punct p2(5, 10); // Apelul constructorului parametrizat
Punct p3 (p2); // Apelul constructorului de copiere

p1.afisare();
p2.afisare();
p3.afisare();
```

```
28 return 0;
29 }
```

4.1.3 Destructorul

Un **destructor** este o funcție membră specială care este apelată automat atunci când un obiect își încheie ciclul de viață (de exemplu, la ieșirea dintr-un bloc de cod sau la eliberarea memoriei alocate dinamic). Destructorii au același nume ca și clasa, precedat de caracterul ~(thilda), și nu au parametri sau tip de returnare.

Exemplu de destructor:

```
1  class Punct {
2  private:
3  int x, y;
4  public:
5  Punct(int initX, int initY) : x(initX), y(initY) {}
6  ~Punct() {
7  std::cout << "Destructor apelat pentru Punct(" << x << ", " << y
       ↪ << ")\n";
8  }
9  };
10
11 int main() {
12 Punct p(3, 4); // Se creeaza un obiect Punct
13 return 0; // La sfarsitul programului, destructorul este apelat
       ↪ automat
14 }
```

Destructorul este util pentru eliberarea resurselor alocate dinamic (de exemplu, memorie alocată cu `new`, fișiere deschise etc.).

4.1.4 Autoreferinţa. Cuvântul cheie “this”

În definiţiile funcţiilor membre sunt necesare referiri la datele membre ale clasei respective. Acest lucru se poate face fără a specifica prin nume un anume obiect. La apelarea unei funcţii membre, identitatea obiectului asupra căruia se acţionează este cunoscută datorită transferului unui parametru implicit catre functiile membre care reprezintă adresa obiectului (pointer) pentru care se executa functia.

Referirea acestei adrese se realizează prin cuvântul cheie **this**, care rerezinta, deci, un pointer către obiectul pentru care s-a apelat funcţia.

Clasa Punct definită anterior se poate completa cu funcţia membră cu prototipul :

```
1 void adresa();
```

definită astfel:

```
void Punct::adresa()
{
    cout<<"\n Adresa obiectului Punct este:";
    cout<< this;
}
```

4.2 Definirea claselor în Python

Clasele în Python oferă o modalitate de a împacheta datele și funcționalitatea împreună. Crearea unei clase implică definirea constructorului acesteia, a atributelor (proprietăților) și a metodelor. Următoarea este o structură generică pentru o clasă în Python:

Exemplu 4.4 Structura generică pentru o clasă în Python

```
class NumeClasa:
    # Constructorul clasei (initializer)
    def __init__(self, atribut1, atribut2):
        # Initializarea atributelor
        self.atribut1 = atribut1
        self.atribut2 = atribut2

    # Metoda pentru a efectua o actiune
    def metoda1(self):
        # Implementarea metodei
        print(f"Atribut1 este {self.atribut1}")

    # Metoda care utilizeaza atributele
    def metoda2(self, parametru1):
        # Implementarea metodei care poate modifica atributele
        self.atribut2 += parametru1
        return self.atribut2


if __name__ == "__main__":
    # Crearea unei instante a NumeClasa
    obiect = NumeClasa("valoare1", 10)

    # Accesarea unei metode
    obiect.metoda1()

    # Modificarea si accesarea unui atribut prin intermediul unei
    ↪ metode
```

```
28      print(obiect.metoda2(5))
```

La rularea exemplului 4.4, efectul afisat în consola este urmatorul:

```
Atribut1 este valoare1
15
```

Această structură descrie o clasă numită *NumeClasa* cu următoarele componente:

- **constructor** (`__init__`): inițializează atributele obiectului. Este apelat atunci când o instanță (obiect) a clasei este creată, folosind numele clasei și parametrii necesari.
- **atribute**: variabile ce aparțin clasei. În structura de mai sus, `atribut1` și `atribut2` sunt definite în constructor, făcându-le atribute ale instanței (fiecare obiect al clasei are propria sa copie).
- **metode**: funcții definite în interiorul unei clase care descriu comportamentele obiectelor clasei. `metoda1` și `metoda2` sunt exemple care operează asupra atributelor obiectului. Aceste metode pot accesa și modifica atributele clasei.
- **crearea unei instanțe**: o instanță a clasei este creată apelând numele clasei urmat de argumentele pe care metoda `__init__` le necesită, așa cum se arată cu `obiect = NumeClasa("valoare1", 10)`.
- **accesarea metodelor și a atributelor**: se utilizează notația cu punct pentru a apela metode (`obiect.metoda1()`) sau pentru a accesa atribute (`obiect.atribut1`) ale instanței clasei.
- `self` reprezintă instanța curentă a clasei. este folosit pentru a accesa variabilele care aparțin clasei. El nu este un cuvânt rezervat în Python, dar este o convenție larg acceptată pentru a se referi la instanța curentă a obiectului. Prin urmare, atunci când definim metode într-o clasă, primul parametru al fiecărei metode este întotdeauna `self`, care este o referință la instanța curentă a clasei. Acest lucru permite metodelor să acceseze și să modifice starea obiectului sau să apeleze alte metode ale aceleiași instanțe.

Plecând de la structura generală a unei clase, mai jos este un exemplu particularizat pentru classa Punct

Exemplu 4.5 Clasa Punct în Python

```
1  class Punct:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def muta(self, deltaX, deltaY):
7          self.x += deltaX
8          self.y += deltaY
9
10     def locatie(self):
```

```
11          return (self.x, self.y)
```

Această clasă `Punct` are o metodă inițializatoare, '__init__', care stabilește valorile inițiale pentru coordonatele 'x' și 'y'. Include, de asemenea, metode pentru a muta punctul și pentru a raporta locația sa actuală.

4.2.1 Funcția `__init__()`

Funcția `__init__()` în Python este una dintre funcțiile speciale, cunoscută și sub numele de metodă constructor. Această metodă este apelată automat atunci când o nouă instanță a unei clase este creată. Scopul principal al `__init__()` este de a inițializa noile obiecte imediat după ce au fost create, atribuindu-le valorile inițiale ale atributelor obiectului.

Sintaxa standard a funcției `__init__()` include cuvântul cheie `def` urmat de `__init__()`, care este urmat de o listă de parametri între paranteze. Primul parametru este întotdeauna `self`, care reprezintă instanța clasei. Alți parametri pot fi adăugați după `self` pentru a transmite valori la inițializarea obiectului.

```
1 class NumeClasa:
2     def __init__(self, parametru1, parametru2):
3         self.atribut1 = parametru1
4         self.atribut2 = parametru2
```

Următorul exemplu demonstrează crearea unei clase simple cu o funcție `__init__()`, care inițializează două atribute ale clasei.

```
1  class Masina:
2      def __init__(self, marca, model):
3          self.marca = marca
4          self.model = model
5
6  # Crearea unei instante a clasei Masina
7  masina_mea = Masina("Toyota", "Corolla")
8
9  # Accesarea atributelor
10 print(f"Marca masinii mele este {masina_mea.marca} si modelul este
   ↪ {masina_mea.model}.")
```

4.2.2 Funcția `__str__()`

Metoda specială `__str__()` în Python este utilizată pentru a returna reprezentarea unui obiect sub formă de șir de caractere. Este apelată de funcții precum `print()` pentru a afișa o reprezentare "frumoasă", ușor de citit, a obiectului. În absența unei metode `__str__()` definite în clasă, se va folosi reprezentarea standard, care adesea nu este foarte informativă.

Sintaxa pentru definirea metodei `__str__()` este similară cu alte metode ale clasei. Primul și singurul parametru obligatoriu este `self`, care reprezintă instanța curentă a clasei.

```
class NumeClasa:
    def __str__(self):
        return "Text care descrie instanta"
```

Următorul exemplu ilustrează cum se poate defini și utiliza metoda `__str__()` pentru a îmbunătăți modul în care obiectele sunt reprezentate sub formă de șiruri de caractere.

```
class Masina:
    def __init__(self, marca, model):
        self.marca = marca
        self.model = model

    def __str__(self):
        return f"Masina {self.marca} {self.model}"

# Crearea unei instante a clasei Masina
masina_mea = Masina("Toyota", "Corolla")

# Utilizarea metodei __str__()
print(masina_mea)
```

Definirea metodei `__str__()` într-o clasă permite dezvoltatorilor să controleze modul în care obiectele sunt reprezentate sub formă de șiruri de caractere, făcând astfel debug-ul și logarea mai ușoare și mai intuitive. Aceasta este deosebit de utilă când obiectele clasei trebuie să fie afișate într-un format citibil de om sau când se dorește o descriere specifică și concisă a instanțelor clasei.

4.2.3 Parametrul `self`

În Python, `self` reprezintă instanța curentă a clasei și este utilizat pentru a accesa variabilele și metodele asociate acelei instanțe. Prin convenție, `self` este întotdeauna primul parametru al oricărei metode din cadrul unei clase, dar nu este un cuvânt rezervat în Python; puteți utiliza orice alt nume în locul lui `self`, deși acest lucru nu este recomandat din motive de claritate și consistență.

Utilizarea `self` este esențială pentru a distinge între atributelor și metodele de clasă și cele ale instanțelor. Acesta permite o metodă să acceseze alte metode și proprietăți ale aceleiași instanțe, precum și să modifice starea instanței.

```
class ExempluClasa:
    def __init__(self, valoare):
        self.atribut = valoare

    def metoda_instanta(self):
        print(f"Valoarea atributului este {self.atribut}")
```

```
7
8  # Crearea unei instante si apelarea unei metode
9  obiect = ExempluClasa(10)
10 obiect.metoda_instanta()
```

În exemplul de mai sus, `self.atribut` face referire la atributul `atribut` al instanței curente. La fel, `self.metoda_instanta()` ar putea fi folosită în interiorul altei metode a clasei pentru a apela metoda `metoda_instanta` a aceleiași instanțe.

4.3 Crearea obiectelor în Python

Obiectele sunt instanțe ale claselor. Se poate creea un obiect apelând numele clasei urmat de valorile inițiale între paranteze.

Plecând de la exemplul 4.5, mai jos se regăseste sintaxa pentru crearea si utilizarea unui obiect din clasa punct:

```
1  # Crearea unui obiect al clasei Punct
2  punct1 = Punct(10, 20)
3
4  # Mutarea punctului
5  punct1.muta(5, -2)
6
7  # Afisarea locatiei punctului
8  print(punct1.locatie())
```

O clasă în Python oferă flexibilitate în gestionarea atributelor sale, permițând modificarea și ștergerea acestora din instanțele sale. Următoarele exemple demonstrează cum să modificăm și să ștergem atribute ale clasei `Punct`. Pentru a modifica valorile atributelor unei instanțe, putem direct să asignăm noi valori atributelor prin intermediul instanței.

```
1  # Presupunand ca avem o clasa Punct definita astfel:
2  class Punct:
3      def __init__(self, x, y):
4          self.x = x
5          self.y = y
6
7  # Crearea unei instante a clasei Punct
8  p = Punct(1, 2)
9
10 # Modificarea valorilor atributelor
11 p.x = 10
12 p.y = 20
```

Atributele unei instanțe pot fi șterse folosind cuvântul cheie `del`. După ștergere, accesarea atributului va genera o eroare, cu exceptia situatiei in care acesta este redefinit.

```
# Stergerea unui atribut
del p.x

# Incercarea de accesare a atributului x va genera o eroare
# print(p.x)  # AttributeError: 'Punct' object has no attribute 'x
    ↪ '
```

Aceste operații demonstrează flexibilitatea OOP în Python, permițând modificarea dinamică a instanțelor clasei. Este important de menționat că, deși aceste operații sunt importante, ar trebui folosite cu prudență pentru a menține integritatea și coerența datelor în cadrul aplicațiilor.

5. Conceptul de moștenire

Agregarea sau compunerea (ierarhia de obiecte)
Moştenirea
Ierarhii de clase
Moştenirea multiplă

Unul din avantajele programării orientate obiect (POO) îl constituie reutilizarea codului scris. În procesul de reutilizare, în principal prin abstractizare, se pot realiza ierarhii, construite în una din variantele:

- Ierarhie de obiecte (relaţie de tip eng. "part of"), numită tradițional agregare sau compunere de clase;
- Ierarhie de clase (relaţie de tip eng. "is a"), numită tradițional moștenire.

Agregarea sau compunerea (ierarhia de obiecte) este relaţia între două obiecte în care unul dintre obiecte aparţine celuilalt obiect. Semantic, agregarea indică o relaţie de tip "part of" ("parte din").

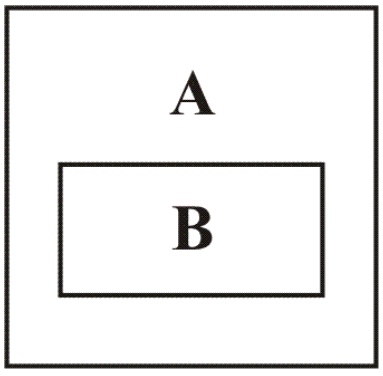

Figura 5.1 Compunerea claselor

În Figura 5.1, obiectul B este parte componentă a obiectului A. Spre exemplu, carcasa este parte componentă a unui vehicul.

Moştenirea (ierarhia de clase) este un principiu aplicat de limbajele de programare orientate pe obiecte care permite refolosirea codului scris şi extinderea funcţionalităţii claselor existente. Între două clase pot exista multe asemănări, dar şi multe diferenţe. Este posibil ca

informaţia comună unor clase să fie descrisă o singură dată. Mecanismul moştenirii permite crearea unei ierarhii de clase şi trecerea de la clasele generale la clase particulare. Procesul presupune definirea la început a unei clase, numită clasă de bază, care stabileşte atributele comune tuturor obiectelor ce vor deriva din clasa de bază. Prin moştenire, un obiect poate prelua proprietăţile obiectelor din clasa de bază.

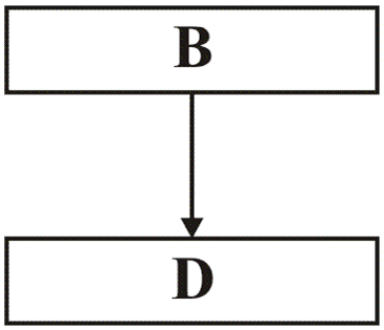

Figura 5.2 Relaţia clasă de bază- clasă derivată

În Figura 5.2, clasa B reprezintă clasa de bază și conține informațiile comune, în timp ce clasa D reprezintă clasa derivată care extinde funcționalitatea clasei de bază.

Clasa B reprezintă *clasa de bază* (este o generalizare) şi conţine informaţiile comune (disponibile prin moştenire şi subclaselor acesteia).

Clasa D reprezintă *clasa derivată* (este o particularizare, o specializare a clasei A) care extinde funcţionalitatea clasei de bază şi conţine informaţiile specifice.

Să presupunem că **B** reprezintă clasa vehiculelor (cu proprietăţile caracteristice: putere, greutate, dimensiuni, etc.), iar **D** reprezintă clasa vehiculelor nautice. În momentul definirii clasei derivate **D**, aceasta moşteneşte toate caracteristicile clasei **B**, rămânând de specificat doar trăsăturile distinctive.

În acest caz, **B** este *clasă de bază*, iar **D** *clasă derivată* (subclasă a clasei **B**), sau **D** este clasă, iar **B** este o *superclasă* a clasei **D**.

Moştenirea poate fi:

- Moştenire unică (simplă) - fiecare clasă are doar o superclasă (vezi 5.3)
- Moştenire multiplă – o clasă are mai multe superclase, rezultând o structură de reţea (vezi 5.4)

În cazul **moștenirii unice**, specializarea clasei se face prin:

- Introducerea de noi atribute și noi metode în clasa derivată (particulare doar clasei derivate).

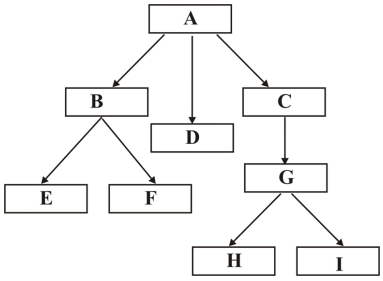

Figura 5.3 A Moștenire unică

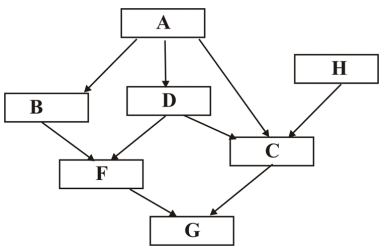

Figura 5.4 Moștenire multiplă

- Redefinirea membrilor în clasa derivată (*polimorfism*).

Moștenirea multiplă este utilă, dar poate crea ambiguități, aceeași membri pot fi moșteniți în mod repetat. Există diverse strategii prin care se pot evita astfel de situații, de exemplu, folosirea claselor virtuale.

5.1 Agregarea sau compunerea (ierarhia de obiecte)

Clasele pot include membri de orice tip, cu excepția tipului de clasă pe care îl definesc, astfel încât membrii unei clase pot fi, la rândul lor, de un tip definit printr-o altă clasă. Clasele care conțin membri de tip clasă sunt denumite clase compuse.

De exemplu, dacă este declarată clasa `tablou` pentru reprezentarea unui tablou unidimensional de dimensiune oarecare, se poate declara o clasă `elev` care să conţină ca membrii numele unui elev și un membru de tip `tablou` pentru înregistrarea notelor acestuia.

În limbajul C++, declaraţia clasei `elev` poate fi:

```
1  class elev
2  {
3      char nume[30];
4      tablou note; // membru de tip tablou
5  public: elev();
6      ~elev();
7      void citire_nume();
8      void citire_note();
9      void modif_nota(int, double);
10     void afisare(); // afiseaza numele, notele si media elevului
11 };
```

Echivalent, in limbajul Python, declaraţia clasei `elev` poate fi:

```
1  class Elev:
2      def __init__(self, nume='', note=None):
3          if note is None:
4              note = [0] * 8  # Default list of 8 zeros
5          self.nume = nume
6          self.note = note
7
8      def citire_nume(self):
9          pass
10
11     def citire_note(self):
12         self
13
14     def modif_nota(self, index, noua_nota):
15         self
```

```
16
17      def afisare(self):
18          self
```

Când se creează un obiect de tip `elev`, pentru inițializarea membrului `note`, în absența unui constructor declarat, va fi utilizat constructorul generat automat de compilator. Dacă în clasa `tablou` există un constructor declarat, atunci la crearea membrului `note` se va apela constructorul respectiv. În cazul în care acest constructor este unic și are parametri ordinari, valorile corespunzătoare acestora trebuie transmise, ceea ce face ca un constructor implicit generat de compilator sau un constructor implicit declarat pentru clasa care conține membrul obiect să nu poată rezolva situația. Este necesar fie să se definească un constructor implicit pentru clasa membrului, fie să se creeze un constructor care să transfere parametrii către constructorul clasei membrului înglobat, așa cum se ilustrează în exemplul următor.

În limbajul C++:

```
1 elev::elev(int nr_note, char * sir) : note(nr_note)
2 { strcpy(nume, sir); }
```

În limbajul Python:

```
1 class Elev:
2     def __init__(self, nr_note, sir):
3         self.note = [0] * nr_note  # Initializeaza o lista cu zero
            ↪ -uri de lungimea nr_note
4         self.nume = sir  # Atribuirea directa functioneaza
            ↪ deoarece sirurile in Python sunt imuabile
```

La definirea constructorului `elev`, în antetul funcţiei, s-a precizat care dintre parametrii este transferat constructorului `note`. Se apelează întâi constructorul `tablou()` care creează membrul "note" şi apoi constructorul `elev()`. Ordinea eliminării obiectelor este inversă creării lor, întâi se distruge obiectul `elev` şi apoi `note`. În exemplele 5.1 și 5.2 este prezentată declaraţia claselor `tablou` şi `elev`, cu un minim de membrii, date şi funcţii, necesare exemplificării aspectelor analizate.

În limbajul C++:

```
1 void elev::citire_nume() // citirea numelui de la tastatura
2 {
3     cout << "\nIntroduceti numele elevului:";
4     cin >> nume;
5 }
```

În limbajul Python:

```
1 class Elev:
2  def citire_nume(self):
```

```
3         # Citirea numelui de la tastatura
4         self.nume = input("\nIntroduceti numele elevului: ")
```

Funcţiile `citire_note()`, `modif_nota(int, double)` şi `afisare()` fac referire la conţinutul membrului `note`, adică la membrii clasei `tablou`. Având în vedere faptul că în clasa `tablou` datele sunt declarate cu acces private, ele nu pot fi referite direct prin expresii de genul `note.nr_el`, `note.tab` în funcţiile membre ale clasei `elev`. Această problemă poate fi rezolvată prin folosirea funcţiilor membre ale clasei `tablou` care sunt cu acces public. De exemplu, pentru citirea notelor se poate defini funcţia `citire_note()` astfel:

În limbajul C++:

```
1 void elev::citire_note() // citirea notelor elevului
2 {
3     cout << "\nIntroduceti notele elevului " << nume << endl;
4     note.citire();
5 }
```

În limbajul Python:

```
1  class Elev:
2      def citire_note(self):
3              # Citirea notelor elevului
4              print(f"\nIntroduceti notele elevului {self.nume}:")
5              while True:
6                  nota = input("Introduceti o nota (sau lasati gol
                       ↪ pentru a termina): ")
7                  if nota == '':
8                      break  # Inceteaza citirea daca inputul este
                           ↪ gol
9                  try:
10                     self.note.append(float(nota))  # Adauga nota
                           ↪ convertita la float in lista de note
11                 except ValueError:
12                     print("Va rugam introduceti un numar valid.")
```

De crearea şi distrugerea membrului `note` se ocupă constructorii, respectiv destructorul clasei `tablou`. Ne existând alte alocări de memorie în clasa `elev`, practic nu este necesară declararea destructorului `elev`. Se observă că în exemplele 5.1 și 5.2 acest destructor nu conţine nici o instrucţiune, el fiind introdus doar pentru sublinierea aspectului semnalat.

Exemplu 5.1 Definitia, în C++, a clasei elev pentru reprezentarea datelor unui elev

```
1 /*
2 Definirea de clase cu membrii obiecte.
3
```

```
4  Se defineste clasa elev pentru reprezentarea datelor unui elev, in
   ↪  care notele elevului sunt gestionate printr-un camp de tip
   ↪ tablou.
5
6  */
7
8  #include <iostream>
9  #include <string.h>
10 #include <conio.h>
11
12 class tablou
13 {    int nr_el; // dimensiunea tabloului tab
14      double *tab; // adresa tabloului tab
15       public:
16      tablou(int=8); // constructor cu parametru implicit
17      tablou(tablou &);   // constructor de copiere
18      ~tablou(); // destructor
19      void citire();
20      void afisare();
21      void modif_el(int, double);
22      double media();
23 };
24
25 class elev
26 {
27      char nume[30];
28      tablou note;
29 public:
30      elev(int=8, char * =" ");  // constructor cu parametrii
          ↪ impliciti - primul
31              // determina numarul de note, deci dimensiunea
32              // tabloului, al doile numele elevului
33      ~elev(); // destructor
34      void citire_nume(); // citirea numelui de la tastatura
35      void citire_note();  // citirea notelor de la tastatura
36      void modif_nota(int, double); // modificarea unei note
37      void afisare(); // afisarea datelor elevului
38 };
39
40 int main()
41 {
```

```
42    elev e1(6,"Ionescu"); // se declara un obiect de tip elev
43    e1.citire_note();
44    e1.afisare();
45    std::cout<<"\nSe modifica o nota a elevului";
46    e1.modif_nota(2, 5.5);
47    e1.afisare();
48    getch();
49    return 1;
50  }
51
52  tablou::tablou(int n) // constructor cu parametru
53  {
54     nr_el=n;
55     tab=new double[nr_el];
56     for (int i=0; i<nr_el; i++)
57          tab[i]=0;
58  }
59
60  tablou::tablou(tablou& t) // constructor de copiere
61  {
62    nr_el=t.nr_el;
63    tab=new double[nr_el];
64    for (int i=0; i<nr_el; i++)
65          tab[i]=t.tab[i];
66  }
67
68  tablou::~tablou()    // destructor
69  {
70     delete [] tab;
71  }
72
73  void tablou::citire() // citirea elementelor tabloului
74  {
75     for(int i=0; i<nr_el; i++)
76      {
77          std::cout << "tab[" << i << "]=";
78          std::cin >> tab[i];
79      }
80  }
81
82  void tablou::afisare() // afisarea elementelor tabloului
```

```
83 {
84     std::cout << std::endl;
85     for(int i=0; i<nr_el; i++)
86         std::cout << "tab[" << i << "]=" << tab[i] <<std::endl;
87 }
88
89
90 void tablou::modif_el(int i, double val) // modificarea unui
   ↪ element al tabloului
91 {
92     if (i<nr_el)
93         tab[i]=val;
94 }
95
96 double tablou::media() // calculul mediei aritmetice a elementelor
   ↪  tabloului
97 {
98   double med=0;
99   for(int i=0; i<nr_el; i++)
100         med += tab[i];
101   med/=nr_el;
102   return med;
103 }
104
105 elev::elev(int nr_note, char * sir):note(nr_note) // constructor
   ↪ cu parametri
106 {
107    strcpy(nume, sir);
108 }
109
110 elev::~elev()  // destructor
111 {
112    //destructorul este formal;
113    //destructorul pentru note se apeleaza in mod implicit
114 }
115
116
117 void elev::citire_nume()     // citirea numelui de la tastatura
118 {
119   std::cout << "\nIntroduceti numele elevului:";
120   std::cin >> nume;
```

```
121 }
122
123 void elev::citire_note() // citirea notelor elevului
124 {
125   std::cout << "\nIntroduceti notele elevului "<< nume <<std::endl
      ↪  ;
126   note.citire();
127 }
128
129 void elev::modif_nota(int i, double val) // modificarea unei note
    ↪ a elevului
130 {
131   note.modif_el(i, val);
132 }
133
134 void elev::afisare()      // afisarea datelor elevului
135 {
136   std::cout << "\nNumele elevului: " << nume;
137   std::cout << "\nNotele:";
138   note.afisare();
139   std::cout << "\nMedia:" << note.media() << std::endl;
140 }
```

Exemplu 5.2 Definitia, în Python, a clasei elev pentru reprezentarea datelor unui elev

```
1  class Tablou:
2      def __init__(self, n=8):
3          self.nrel = n
4          self.tab = [0.0] * self.nrel  # Initialize list of zeros
5
6      def citire(self):
7          # Citirea elementelor tabloului
8          for i in range(self.nrel):
9              self.tab[i] = float(input(f"tab[{i}]="))
10
11     def afisare(self):
12         # Afisarea elementelor tabloului
13         for i in range(self.nrel):
14             print(f"tab[{i}]={self.tab[i]}")
15
16     def modif_el(self, i, val):
17         # Modificarea unui element al tabloului
```

```
18          if i < self.nrel:
19              self.tab[i] = val
20
21      def media(self):
22          # Calculul mediei aritmetice a elementelor tabloului
23          return sum(self.tab) / self.nrel
24
25
26  class Elev:
27      def __init__(self, nr_note=8, sir=''):
28          self.nume = sir
29          self.note = Tablou(nr_note)
30
31      def citire_nume(self):
32          # Citirea numelui de la tastatura
33          self.nume = input("\nIntroduceti numele elevului:")
34
35      def citire_note(self):
36          # Citirea notelor elevului
37          print(f"\nIntroduceti notele elevului {self.nume}:")
38          self.note.citire()
39
40      def modif_nota(self, i, val):
41          # Modificarea unei note a elevului
42          self.note.modif_el(i, val)
43
44      def afisare(self):
45          # Afisarea datelor elevului
46          print(f"\nNumele elevului: {self.nume}")
47          print("\nNotele:")
48          self.note.afisare()
49          print(f"\nMedia: {self.note.media()}")
50
51  #Partea Main a scriptului
52  if __name__ == "__main__":
53      e1 = Elev(6, "Ionescu")  # Se declara un obiect de tip elev
54      e1.citire_note()
55      e1.afisare()
56      print("\nSe modifica o nota a elevului")
57      e1.modif_nota(2, 5.5)
58      e1.afisare()
```

5.2 Moştenirea (ierarhia de clase)

Unul dintre principiile fundamentale ale programării orientate pe obiecte este moștenirea. Prin intermediul moștenirii, se creează noi seturi de clase care extind o clasă de bază deja definită, adăugând noi proprietăți acesteia.

Procesul de moștenire are ca scop obținerea unor clase derivate, care moștenesc caracteristicile unei clase de bază preexistente. Clasele derivate includ toți membrii clasei de bază, la care se adaugă noi membri, date și funcții. Clasa de bază rămâne neschimbată în urma derivării și poate fi compilată înainte de a se efectua derivările.

O clasă de bază poate fi extinsă pentru a deveni la rândul său bază pentru alte clase derivate, iar această ierarhie continuă formând o structură ierarhică de clase.

Începând cu clase simple și generale, fiecare nivel al ierarhiei adaugă noi proprietăți, rezultând clase tot mai complexe și specializate. Tehnica derivării contribuie la creșterea productivității în procesul de programare. De asemenea, este posibil să se definească clase derivate care combină mai multe clase de bază, proces cunoscut sub denumirea de moștenire multiplă.

5.2.1 Declararea unei clasei derivate

Sintaxa generală pentru declararea unui tip de date `class` ca fiind clasă de bază, respectiv pentru derivarea unei clase, în limbajul Python, are următoarea formă:

```
class ClasaDeBază:
    # Implementarea clasei de bază
    pass

class ClasaDerivată(ClasaDeBază):
    # Implementarea clasei derivate
    pass
```

În cazul moștenirii simple, ClasaDerivată moștenește toate metodele și atributele din ClasaDeBază.

Un echivalent al sintaxei generale pentru declararea unui tip de date `class` în C++, care include și posibilitatea de derivare, este următoarea:

```
class <nume_clasa> <: lista_clase_baza> {<lista_membri>} <lista_variabile>;
```

unde lista_clase_baza conține una (în cazul unei derivări simple) sau mai multe (în cazul unei derivări multiple) clase de bază, fiecare fiind urmată de un specificator de acces. Acest specificator este specific limbajului C++ și nu se regăsește în Python. În C++, specificatorul de acces controlează drepturile de acces la membrii clasei de bază. Prin urmare, lista claselor de bază are următoarea sintaxă:

Tabela 5.1 Atributele de acces moștenite de clasele derivate

Atributul din clasa de bază	**Modificatorul de acces**	**Accesul moștenit de clasa derivată**	**Accesul din exterior**
private	private	inaccesibil	inaccesibil
protected		private	inaccesibil
public		private	inaccesibil
private	public	inaccesibil	inaccesibil
protected		protected	inaccesibil
public		public	accesibil

```
specificator_acces clasa_baza_1, specificator_acces clasa_baza_2, ...
```

Specificatorii de acces care pot fi utilizați sunt `public` și `private`, iar valoarea implicită este `private`.

Atributele de acces moștenite de clasa derivată, în funcție de specificatorul de acces folosit, sunt prezentate în Tabelul 5.1.

Pentru a avea acces la membrii clasei de bază dintr-o clasă derivată, aceștia trebuie să fie declarați ca `protected` sau `public`. Pentru a păstra dreptul de acces la membrii clasei de bază, se utilizează specificatorul de acces `public`. Stabilirea specificatorilor de acces trebuie să țină cont de perspectivele de dezvoltare ale ierarhiei de clase, fără a încalca principiul încapsulării datelor. Moștenirea este similară cu procesul de includere a obiectelor în alte obiecte, denumit **agregare sau compunere**. Există câteva caracteristici esențiale ale moștenirii:

- codul poate fi reutilizat de mai multe clase;
- clasele pot fi extinse fără a fi necesară recompilarea claselor originare;
- funcțiile care utilizează obiecte din clasa de bază pot lucra și cu obiecte din clasele derivate.

Pentru compararea celor două procedee, compunere, respectiv derivare, se consideră clasa `punct` definită pentru reprezentarea punctelor în plan prin coordonate carteziene (folosită în exemplificări ale diferitelor aspecte comentate în capitolul 4) și clasa `cerc`, definită pentru reprezentarea cercurilor descrise prin coordonatele centrului și rază.

În Exemplul 5.3, folosind limbajul C++, procedeul folosit pentru declararea clasei `cerc` este cel de compunere, clasa având ca date membre `raza`, de tip `float` și `centru`, de tip `punct`.

Exemplu 5.3 Exemplu, în limbaj C++, de compunere pentru clasa cerc

```
1 #include <iostream>
2
3 class punct
4 {
5     float x, y;
6 public:
```

```
7       punct(float a=0, float b=0)
8       {
9           x = a;
10          y = b;
11      }
12      void modific_punct(float dx, float dy)
13      {
14          x += dx;
15          y += dy;
16      }
17      void afisare()
18      {
19          std::cout << "\nPunct: x=" << x << "\ty=" << y;
20      }
21  };
22
23  class cerc
24  {
25      punct centru;
26      float raza;
27  public:
28      cerc(float r=0)
29      {
30          raza = r;
31      }
32
33      void afisare()
34      {
35          centru.afisare(); // afisarea valorilor corespunzatoare
                ↪ centrului, x si
36          // respectiv y, se face prin functia membra clasei punct;
37          // nu este permis acces direct deoarece membrii x si y
38          // sunt private
39          std::cout << "\tRaza=" << raza;
40      }
41
42      void modific_cerc(float dx, float dy, float dr)
43      {
44          centru.modific_punct(dx, dy); // modificarea valorilor
                ↪ corespunzatoare
45          // centrului se face prin functia membra clasei
```

```
46          // punct; nu este permis acces direct deoarece
47          // membrii x si y sunt private
48          raza += dr;
49      }
50  };
51
52  int main()
53  {
54      cerc c1;
55      c1.modific_cerc(1.1, 2.2, 3.3);
56      c1.afisare();
57      return 0;
58  }
```

La rularea exemplului 5.3, efectul afisat în consola este urmatorul:

```
Punct: x=1.1    y=2.2   Raza=3.3
```

Accesul la membrii privați ai clasei `punct` din funcțiile membre ale clasei `cerc` se realizează exclusiv prin intermediul funcțiilor membre publice ale clasei `punct`.

Dacă s-ar dori acces direct la toți membrii clasei `punct`, aceștia ar trebui declarați cu acces `public`, însă acest lucru ar compromite controlul asupra acestor membri, încălcând astfel principiul încapsulării.

O alternativă ar fi declararea unei relații de prietenie între cele două clase, ceea ce se poate realiza prin completarea declarației clasei `punct` astfel:

```
1  class punct
2  {
3  ....
4  friend class cerc;
5  ...
6  };
```

Dezavantajul îl constituie faptul că este necesară modificarea declarației clasei `punct`.

Omolog, în Exemplul 5.4, folosind limbajul Python, procedeul folosit pentru declararea clasei `cerc` este cel de compunere, clasa având ca date membre `raza`, și `centru` de tip `punct` .

Exemplu 5.4 Exemplu, în limbaj Python, de compunere pentru clasa cerc

```
1  class Punct:
2      def __init__(self, a=0, b=0):
3          self.x = a
4          self.y = b
5
```

```
6       def modific_punct(self, dx, dy):
7           self.x += dx
8           self.y += dy
9
10      def afisare(self):
11          print(f"\nPunct: x={self.x}\ty={self.y}")
12
13
14  class Cerc:
15      def __init__(self, r=0):
16          self.centru = Punct()
17          self.raza = r
18
19      def afisare(self):
20          self.centru.afisare()  # Afisarea valorilor
                ↪ corespunzatoare centrului, x si
21          # respectiv y, se face prin functia membra clasei Punct;
22          # nu este permis acces direct deoarece membrii x si y
23          # sunt private
24          print(f"\tRaza={self.raza}")
25
26      def modific_cerc(self, dx, dy, dr):
27          self.centru.modific_punct(dx, dy)  # Modificarea valorilor
                ↪  corespunzatoare
28          # centrului se face prin functia membra clasei
29          # Punct; nu este permis acces direct deoarece
30          # membrii x si y sunt private
31          self.raza += dr
32
33
34  def main():
35      c1 = Cerc()
36      c1.modific_cerc(1.1, 2.2, 3.3)
37      c1.afisare()
38
39
40  if __name__ == "__main__":
41      main()
```

La rularea exemplului 5.4, efectul afisat în consola este urmatorul:

```
Punct: x=1.1    y=2.2
```

```
Raza=3.3
```

Referitor la procedeul de derivare, în Exemplul 5.5 se definește clasa `cerc` prin derivare din clasa `punct`.

Exemplu 5.5 Exemplu, în limbaj C++, pentru definirea clasei cerc prin derivare din clasa punct.

```
1  #include <iostream>
2
3  class punct
4  {
5    protected:  // se foloseste specificatorul de acces protected
       ↪ pentru a permite
6    // accesul la datele membre ale clasei punct din clasele
       ↪ derivate,
7    // dar asigurandu-se in continuare protectia fata de functiile
8    // externe acestora
9      float x, y;
10   public:
11     punct(float a=0, float b=0)
12     {
13       x=a;
14       y=b;
15     }
16     void modific_punct(float dx, float dy)
17     {
18       x+=dx;
19       y+=dy;
20     }
21     void afisare()
22     {
23       std::cout<<"\nPunct: x="<<x<<"\ty="<<y;
24     }
25 };
26
27 class cerc : public punct   // in declaratie se specifica clasa de
     ↪  baza punct cu
28 {
29     // acces public
30     float raza;
31   public:
32     cerc(float r=0)
33     {
```

```
34        raza=r;
35      }
36      void afisare()
37      {
38        std::cout<<"\nCentru cerc: x="<<x<<"\ty="<<y; // accesul la
            ↪ membrii clasei
39        // punct se face direct datorita specificatorului protected
            ↪ din clasa punct si a specificatorului public atasat
            ↪ clasei de baza
40        std::cout<<"\tRaza="<<raza;
41      }
42      void modific_cerc(float dx, float dy, float dr)
43      {
44        x+=dx; // accesul la membrii clasei
45        y+=dy; // punct se face direct
46        raza+=dr;
47      }
48  };
49
50  int main()
51  {
52    punct p1(1.5, 2.5);
53    p1.modific_punct(1, 1);
54    p1.afisare();
55    cerc c1;
56    c1.modific_cerc(1.1, 2.2, 3.3);
57    c1.afisare();
58    c1.modific_punct(2.5, 4.6); // accesul la functiile membre
        ↪ clasei punct
59     // prin obiecte de tip cerc este permis
60    c1.afisare();
61    cerc *p_cerc; // declaratia unui pointer la tipul cerc
62    p_cerc = &c1;
63    p_cerc->afisare(); // apelul functiilor prin pointerul p_cerc
64    p_cerc->modific_cerc(1.1, 2.2, 3.3);
65    p_cerc->modific_punct(2.5, 4.6);
66    p_cerc->afisare();
67    return 0;
68  }
```

La rularea exemplului 5.5, efectul afisat în consola este urmatorul:

```
Punct: x=2.5    y=3.5
Centru cerc: x=1.1      y=2.2   Raza=3.3
Centru cerc: x=3.6      y=6.8   Raza=3.3
Centru cerc: x=3.6      y=6.8   Raza=3.3
Centru cerc: x=7.2      y=13.6  Raza=6.6
```

Clasa derivată moștenește toate proprietățile clasei de bază (membrii `x`, `y`, funcțiile `afișare()` și `modific_punct()`), adăugându-le un membru suplimentar, `raza`, funcțiile `cerc()`, `modific_cerc()` și `afișare()`. Se remarcă faptul că funcția `modific_punct()`, membru al clasei de bază, poate fi apelată prin obiecte de tip `cerc`. În varianta din Exemplul 5.3, acest lucru nu este posibil, funcția putând fi accesată doar prin membrul `centru`, care este declarat cu acces `private`.

Obiectele de tip `cerc` și apelul funcțiilor membre pot fi, de asemenea, realizate prin intermediul pointerilor, exemplificarea fiind oferită în Exemplul 5.5.

Este permisă atribuirea obiectelor derivate unor obiecte ale clasei de bază, iar astfel, în funcția `main()` din exemplul anterior, se poate include secvența:

```
1 p1=c1; // p1.x preia valoarea c1.x, iar p1.y preia valoarea c1.y
```

Prin operația de atribuire se copiază doar membrii clasei de bază. Regula de compatibilitate se poate aplica și pentru pointerii și referințele de obiecte.

```
1 punct *p_p;
2 cerc *p_c;
3
4 p_p=p_c;
```

În limbajul Python, procesul de derivare, folosind ca și punct de plecare exemplul omolog din C++, este următorul:

Exemplu 5.6 Exemplu, în limbaj Python, de derivare a clasei cerc din clasa punct

```
1  class Punct:
2      def __init__(self, a=0, b=0):
3          self._x = a  # Echivalentul specificatorului "protected"
               ↪ din C++
4          self._y = b  # Membrii sunt accesibili in clasele derivate
               ↪ , dar nu direct din exterior
5
6      def modific_punct(self, dx, dy):
7          self._x += dx
8          self._y += dy
9
10     def afisare(self):
11         print(f"\nPunct: x={self._x}\ty={self._y}")
```

```
12
13
14 class Cerc(Punct):  # Derivare din clasa Punct cu acces public
15     def __init__(self, r=0):
16         super().__init__()  # Apelarea constructorului clasei de
               ↪ baza (Punct)
17         self.raza = r
18
19     def afisare(self):
20         print(f"\nCentru cerc: x={self._x}\ty={self._y}")  #
               ↪ Accesul la membrii _x si _y este permis
21         print(f"\tRaza={self.raza}")
22
23     def modific_cerc(self, dx, dy, dr):
24         self._x += dx  # Acces direct la membrii clasei de baza
25         self._y += dy
26         self.raza += dr
27
28
29 def main():
30     p1 = Punct(1.5, 2.5)
31     p1.modific_punct(1, 1)
32     p1.afisare()
33
34     c1 = Cerc()
35     c1.modific_cerc(1.1, 2.2, 3.3)
36     c1.afisare()
37
38     c1.modific_punct(2.5, 4.6)  # Acces la functiile clasei de
           ↪ baza prin obiectul de tip Cerc
39     c1.afisare()
40
41     p_cerc = c1  # Atribuirea unui obiect Cerc unei variabile
42     p_cerc.afisare()  # Apelul functiilor prin obiectul de tip
           ↪ Cerc
43     p_cerc.modific_cerc(1.1, 2.2, 3.3)
44     p_cerc.modific_punct(2.5, 4.6)
45     p_cerc.afisare()
46
47
48 if __name__ == "__main__":
```

```
49      main()
```

La rularea exemplului 5.6, efectul afisat în consola este urmatorul:

```
Punct: x=2.5    y=3.5

Centru cerc: x=1.1      y=2.2
        Raza=3.3

Centru cerc: x=3.6      y=6.8
        Raza=3.3

Centru cerc: x=3.6      y=6.8
        Raza=3.3

Centru cerc: x=7.2      y=13.6
        Raza=6.6
```

5.2.2 Constructori şi destructori pentru clasa derivată

Atunci când se creează un obiect, se apelează întotdeauna constructorul corespunzător clasei respective, iar declarația obiectului trebuie să specifice valorile inițiale cerute de parametrii constructorului, în cazul în care aceștia există. Distrugerea obiectului este precedată de apelul funcției destructor ale aceleași clase.

Aceste reguli se aplică și în cazul claselor derivate. La crearea unui obiect al unei clase derivate, se alocă memorie pentru datele membre ale ambelor clase, atât pentru clasa de bază, cât și pentru clasa derivată. Se apelează mai întâi constructorul clasei de bază și apoi constructorul clasei derivate. Dacă constructorii sunt definiți cu parametri, declarația obiectului derivat trebuie să conțină valorile de inițializare atât pentru membrii proprii ai clasei derivate, cât și pentru membrii clasei de bază. Astfel, când se definește constructorul clasei derivate, iar clasa de bază nu dispune de un constructor implicit sau cu parametri impliciți, este necesar să se specifice parametrii care vor fi preluați de constructorul clasei de bază. Această specificare se adaugă la antetul funcției constructor a clasei derivate, folosind, de exemplu în C++, următoarea sintaxă:

```
1 cerc(float a, float b, float r): punct(a, b)
2 {
3     ...
4 }
```

Prin această declarație, se precizează că parametrii `a` și `b` vor fi folosiți pentru inițializarea membrilor clasei `punct`, `x` respectiv `y`.

În Exemplul 5.7 este reluată declarația claselor `punct` și `cerc`, cu definire de constructori și destructori.

Exemplu 5.7 Exemplu, în limbaj C++, pentru declaraţia claselor punct şi cerc, cu definire de constructori şi destructori (plecand de la exemplele anterioare).

```
1  #include <iostream.h>
2
3  class punct
4  {
5    protected:
6        float x, y;
7    public :
8        punct(float a, float b)
9        {
10         x=a; y=b;
11         cout<<"\nConstructor punct: this="<<this;
12         cout<<"\tx="<<x<<"\ty="<<y;
13       }
14
15       ~punct()
16       {
17         cout<<"\nDestructor punct: this="<<this;
18         cout<<"\tx="<<x<<"\ty="<<y<<endl;
19       }
20 };
21
22 class cerc:public punct // se declara clasa cerc derivata din
     ↪ clasa punct
23 {
24       float raza;
25   public:
26       cerc(float a, float b, float r):punct(a, b) // in antetul
           ↪ constructorului se
27      // specifica efectiv care sunt parametrii transferati
          ↪ constructorului punct
28             raza=r;
29             cout<<"\nConstructor cerc: this="<<this;
30             cout<<"\traza="<<r<<endl;
31         }
32       ~cerc()
33         {
34             cout<<"\nDestructor cerc: this="<<this;
35             cout<<"\traza="<<raza;
36         }
```

```
37 };
38
39 void main()
40 {
41     cerc c1(1.1, 2.2, 10), c2(5, 10, 15);
42 }
```

La rularea exemplului 5.7, efectul afisat în consola este urmatorul:

```
Constructor punct: this=0xffea x=1.1 y=2.2
Constructor cerc: this=0xffea  raza=10

Constructor punct: this=0xffde x=5 y=10
Constructor cerc: this=0xffde  raza=15

Destructor cerc: this=0xffde  raza=15
Destructor punct: this=0xffde  x=5 y=10

Destructor cerc: this=0xffea  raza=10
Destructor punct: this=0xffea  x=1.1 y=2.2
```

La definirea constructorului clasei `cerc`, s-a precizat care dintre parametrii transmiși sunt preluați de constructorul clasei `punct`. Au fost adăugate mesaje de afișare pentru a urmări ordinea în care sunt apelate funcțiile. Se observă că, la crearea unui obiect de tip `cerc`, se apelează mai întâi constructorul clasei `punct`, urmat de constructorul clasei `cerc`. De asemenea, se poate observa că adresa obiectului este unică, aceasta fiind asociată tuturor elementelor care formează obiectul `cerc`, astfel încât ambele constructori, `punct()` și `cerc()`, vor afișa aceeași valoare. În ceea ce privește eliminarea obiectelor de tip `cerc`, apelul destructorilor se face în ordine inversă, astfel că mai întâi se va apela destructorul `cerc()` și apoi `punct()`. În cazul în care clasele de bază au definit un constructor implicit sau un constructor cu parametri impliciți, nu este necesar să se specifice parametrii care sunt transmiși obiectului clasei de bază.

Omolog, în Python, exemplul 5.8 reia declarația claselor `punct` și `cerc`, cu definire de constructori și destructori,

Exemplu 5.8 Exemplu, în limbaj Python, pentru declaraţia claselor punct şi cerc, cu definire de constructori şi destructori (plecand de la exemplele anterioare).

```
1 class Punct:
2     def __init__(self, a, b):
3         self.x = a
4         self.y = b
```

```
5           print(f"\nConstructor Punct: this={hex(id(self))}")
6           print(f"\tx={self.x}\ty={self.y}")
7
8       def __del__(self):
9           print(f"\nDestructor Punct: this={hex(id(self))}")
10          print(f"\tx={self.x}\ty={self.y}")
11
12  class Cerc(Punct):  # Clasa Cerc derivata din clasa Punct
13      def __init__(self, a, b, r):
14          # Constructorul clasei Cerc, care apeleaza constructorul
                ↪ clasei Punct
15          super().__init__(a, b)  # Se specifica parametrii
                ↪ transferati constructorului Punct
16          self.raza = r
17          print(f"\nConstructor Cerc: this={hex(id(self))}")
18          print(f"\traza={self.raza}")
19
20      def __del__(self):
21          # Destructorul clasei Cerc
22          print(f"\nDestructor Cerc: this={hex(id(self))}")
23          print(f"\traza={self.raza}")
24
25  def main():
26      c1 = Cerc(1.1, 2.2, 10)  # Crearea primului obiect Cerc
27      c2 = Cerc(5, 10, 15)     # Crearea celui de-al doilea obiect
            ↪ Cerc
28
29  if __name__ == "__main__":
30      main()
```

În urma executiei programului, se va afisa exact același rezultat ca și în exemplul 5.7 .

În python, sintaxa `__init__` în fiecare clasă este constructorul. În clasa Cerc, acesta apelează constructorul clasei de bază Punct folosind super(). Comentariul explică faptul că parametrii sunt transferați către constructorul clasei de bază. Omolog constructorului, sintaxa `__del__` este destructorul fiecărei clase, responsabil pentru curățarea resurselor înainte ca obiectul să fie distrus.

5.2.3 Constructorul de copiere pentru o clasă derivată

În ceea ce privește utilizarea constructorului de copiere pentru o clasă derivată, există mai multe situații care trebuie luate în considerare.

Dacă nici clasa derivată, nici clasa de bază nu au definit un constructor de copiere, se va

apela constructorul implicit generat de compilator. În acest caz, copierea se face membru cu membru (vezi Exemplul 5.9).

Exemplu 5.9 Exemplu, în limbaj C++, pentru declarația constructorului de copiere de compilator pentru o clasă derivată.

```
1  #include <iostream>
2
3  class punct
4  {
5    protected:
6        float x, y;
7    public:
8        punct(float a, float b)
9        {
10           x=a;
11           y=b;
12           std::cout << "\nConstructor punct:";
13           std::cout << "\tx=" << x << "\ty=" << y;
14       }
15
16       ~punct()
17       {
18           std::cout << "\nDestructor punct: ";
19           std::cout << "\tx=" << x << "\ty=" << y << std::endl;
20       }
21 };
22
23 class cerc : public punct
24 {
25     float raza;
26   public:
27     cerc(float a, float b, float r) : punct(a, b)
28     {
29         raza = r;
30         std::cout << "\nConstructor cerc: ";
31         std::cout << "\traza=" << r << std::endl;
32     }
33     ~cerc()
34     {
35         std::cout << "\nDestructor cerc: ";
36         std::cout << "\traza=" << raza;
```

```
37      }
38  };
39
40  int main()
41  {
42      cerc c1(1.1, 2.2, 10);
43      cerc c2(c1);
44  }
```

La rularea exemplului 5.9, efectul afisat în consola este urmatorul:

```
Constructor punct:  x=1.1 y=2.2
Constructor cerc:  raza=10

Destructor cerc:  raza=10
Destructor punct:  x=1.1 y=2.2

Destructor cerc:  raza=10
Destructor punct:  x=1.1 y=2.2
```

Crearea obiectului `c2` a avut loc prin apelul constructorului de copiere generat de compilator. Drept urmare, nu se afișează niciun mesaj asociat apelului constructorului, dar se afișează mesajele corespunzătoare eliminării ambelor obiecte, `c1` și `c2`. În cazul în care clasa de bază are definit un constructor de copiere, dar clasa derivată nu, compilatorul va crea un constructor implicit pentru clasa derivată, care va apela constructorul de copiere al clasei de bază.

În limbajul Python, fiind limbaj interpretat, copierea constructorului nu poate fi realizată de compilator, motiv pentru care copierea trebuie facută explicit prin apelul funcției , `copy`, după cum urmează.

Exemplu 5.10 Exemplu, în limbaj Python, pentru declarația constructorului de copiere pentru o clasă derivată.

```
1  import copy
2
3  class Punct:
4      def __init__(self, a, b):
5          self.x = a
6          self.y = b
7          print(f"\nConstructor Punct:")
8          print(f"\tx={self.x}\ty={self.y}")
9
10     def __del__(self):
```

```
11          print(f"\nDestructor Punct: ")
12          print(f"\tx={self.x}\ty={self.y}")
13
14  class Cerc(Punct):
15      def __init__(self, a, b, r):
16          super().__init__(a, b)
17          self.raza = r
18          print(f"\nConstructor Cerc:")
19          print(f"\traza={self.raza}")
20
21      def __del__(self):
22          print(f"\nDestructor Cerc:")
23          print(f"\traza={self.raza}")
24
25  def main():
26      c1 = Cerc(1.1, 2.2, 10)
27      c2 = copy.copy(c1)  # Copierea obiectului c1 in c2 folosind
        ↪ copy
28
29  if __name__ == "__main__":
30      main()
```

La rularea exemplului 5.10, efectul afisat în consola este urmatorul:

```
Constructor Cerc:
        raza=10

Destructor Cerc:
        raza=10

Destructor Cerc:
        raza=10
```

Sintaxa `copy.copy(c1)` în Python este folosită pentru a crea o copie superficială a obiectului `c1`. Acest lucru este similar cu apelul constructorului de copiere implicit creat de compilator în C++. Atât constructorii, cât și destructorii afișează informațiile relevante pentru a ilustra ordinea apelurilor și valorile membrilor obiectelor. Sintaxa `__del__` este utilizată pentru a simula destructorii din C++, afișând mesajele atunci când obiectele sunt distruse.

Pentru clasa punct din Exemplul 5.9 se poate face o copiere explicită, prin includerea definiției constructorului de copiere din Exemplul 5.11.

Exemplu 5.11 Exemplu, în limbaj C++, pentru declarația constructorului de copiere implicit.

```
1 punct(punct &p)
2 {
3       x = p.x;
4       y = p.y;
5       std::cout << "\nConstructor de copiere punct:";
6       std::cout << "\tx=" << x << "\ty=" << y;
7 }
```

La rularea exemplului 5.11, efectul afișat în consolă este urmatorul:

```
Constructor punct:  x=1.1 y=2.2
Constructor cerc:  raza=10

Constructor de copiere punct:  x=1.1 y=2.2

Destructor cerc:  raza=10
Destructor punct:  x=1.1 y=2.2

Destructor cerc:  raza=10
Destructor punct:  x=1.1 y=2.2
```

Se observă afișarea mesajului asociat apelului constructorului de copiere pentru clasa `punct`, care a fost invocat de constructorul de copiere generat automat de compilator pentru clasa `cerc`.

În cazul în care se definește un constructor de copiere pentru clasa derivată, acesta va fi responsabil pentru transferul valorilor corespunzătoare membrilor clasei de bază. Astfel, pentru clasa `cerc`, se va include un constructor de copiere cu prototipul:

```
1 cerc(cerc &);
```

iar antetul este de forma:

```
1 cerc::cerc(cerc &c) : punct(c.x, c.y)
```

așa cum este arătat în exemplul următor.

```
1 cerc::cerc(cerc &c) : punct(c.x, c.y)
2 {
3     raza=c.raza;
4     cout<<"Constructor de copiere cerc: ";
5     cout<<"raza="<<raza;
6 }
```

La rularea exemplului 5.9 cu adaugarea exemplului de mai sus, efectul afișat în consolă este urmatorul:

```
Constructor punct:  x=1.1 y=2.2
Constructor cerc:  raza=10

Constructor punct:  x=1.1 y=2.2
Constructor de copiere cerc:  raza=10

Destructor cerc:  raza=10
Destructor punct:  x=1.1 y=2.2

Destructor cerc:  raza=10
Destructor punct:  x=1.1 y=2.2
```

Se poate observa că, deși există un constructor de copiere definit pentru clasa de bază, pentru crearea obiectului `punct` se apelează constructorul cu parametri, nu cel de copiere. Lista de parametri care se transmit către obiectul clasei de bază poate fi omisă în cazul în care, pentru clasa de bază, este definit un constructor implicit sau un constructor cu parametri implicați.

În Python se pot adauga metode explicite de copiere a constructorului prin sintaxa `__copy__`, după cum urmează (plecând de la exemplul 5.10)

Exemplu 5.12 Exemplu, în limbaj Python, pentru declarația constructorului de copiere pentru o clasă derivată.

```
1  import copy
2
3  class Punct:
4      def __init__(self, a, b):
5          self.x = a
6          self.y = b
7          print(f"\nConstructor Punct:")
8          print(f"\tx={self.x}\ty={self.y}")
9
10     def __del__(self):
11         print(f"\nDestructor Punct:")
12         print(f"\tx={self.x}\ty={self.y}")
13
14     def __copy__(self):
15         print(f"\nConstructor de copiere Punct:")
16         return type(self)(self.x, self.y)
17
18 class Cerc(Punct):
19     def __init__(self, a, b, r):
```

```
20          super().__init__(a, b)
21          self.raza = r
22          print(f"\nConstructor Cerc:")
23          print(f"\traza={self.raza}")
24
25      def __del__(self):
26          print(f"\nDestructor Cerc:")
27          print(f"\traza={self.raza}")
28
29      def __copy__(self):
30          print(f"\nConstructor de copiere Cerc:")
31          return type(self)(self.x, self.y, self.raza)  # Creeaza o
              ↪ copie directa a lui Cerc
32
33  def main():
34      c1 = Cerc(1.1, 2.2, 10)
35      c2 = copy.copy(c1)  # Utilizarea metodei __copy__ pentru a
          ↪ crea o copie superficiala
36
37  if __name__ == "__main__":
38      main()
```

Metoda `__copy__` în `Punct` definește un constructor de copiere pentru `Punct` care returnează o nouă instanță a clasei `Punct` cu aceleași valori pentru x și y. Metoda `__copy__` în Cerc definește un constructor de copiere pentru Cerc care apelează metoda `__copy__` a clasei de bază `(Punct)` pentru a copia valorile x și y, și apoi creează o nouă instanță de `Cerc` cu aceste valori plus valoarea pentru raza. Sintaxa `copy.copy(c1)` folosește metoda copy din modulul copy pentru a realiza copierea obiectului `c1`. Aceasta va apela metoda `__copy__` definită în `Cerc`.

5.2.4 Supradefinirea funcţiilor membre

În C++, o clasă derivată are acces la membrii clasei de bază care sunt marcați cu specificatorii de acces `protected` sau `public`. De exemplu, dacă pentru clasa `punct` este definită o funcție membră de afişare, aceasta va putea fi utilizată și în cadrul clasei derivate.

```
1  punct::afisare()
2  {
3      cout<<"\\nPunct:   x="<<x<<"  , y="<<y;
4  }
```

şi se defineşte funcţia main():

```
1  void main()
```

```
2 {
3     cerc c(1.1, 2.2, 10);
4     c.afisare();
5 }
```

rezultatul execuţiei acestei secvenţe, în contextul exemplului 5.9, este afişarea:

```
Constructor punct:  x=1.1 y=2.2
Constructor cerc:  raza=10

Punct: x=1.1 y=2.2

Destructor cerc:  raza=10
Destructor punct:  x=1.1 y=2.2
```

Funcția de afișare definită în clasa `punct` este moștenită de clasa `cerc`, dar având în vedere că pentru clasa `cerc` este necesar un nivel suplimentar de informație, se va defini o nouă funcție de afișare specifică acestei clase. Supradefinirea funcțiilor membre ale clasei de bază de către funcții corespunzătoare din clasa derivată este permisă. Astfel, clasa `cerc` va include o funcție membră `void afisare()`.

```
1 void cerc::afisare()
2 {
3     cout<<"\\nCerc:";
4     cout<<"\\ncentru: x="<<x<<"\\ty="<<y;
5     cout<<"\\nraza: "<<raza;
6 }
```

Apelul funcţiei :

```
1 c.afisare();
```

are ca efect afişarea:

```
Cerc:
centru: x=1.1 y=2.2
raza= 10
```

Noile definiţii din clasa derivată nu substituie definiţiile din clasa de bază, ci se adaugă acestora.

În Python, supradefinirea funcțiilor membre ale unei clase de bază într-o clasă derivată se face prin redefinirea acelei funcții în clasa derivată. Python folosește același concept de supradefinire ca și C++, dar sintaxa este diferită. Plecând de la exemplul de mai sus referitor la supradefinirea funcției `afișare`, în Python supradefinirea se realizează după cum este exemplificat în mai jos:

Exemplu 5.13 Exemplu, în limbaj Python, pentru supradefinirea funcției `afișare`

```
1  class Punct:
2      def __init__(self, a=0, b=0):
3          self.x = a
4          self.y = b
5          print(f"\nConstructor punct: x={self.x} y={self.y}")
6
7      def afisare(self):
8          print(f"\nPunct: x={self.x} , y={self.y}")
9
10     def __del__(self):
11         print(f"\nDestructor punct: x={self.x} y={self.y}")
12
13 class Cerc(Punct):
14     def __init__(self, a=0, b=0, r=0):
15         super().__init__(a, b)  # Apelam constructorul clasei de
               ↪ baza
16         self.raza = r
17         print(f"\nConstructor cerc: raza={self.raza}")
18
19     def afisare(self):
20         print(f"\nCerc:")
21         print(f"centru: x={self.x}\ty={self.y}")
22         print(f"raza: {self.raza}")
23
24     def __del__(self):
25         print(f"\nDestructor cerc: raza={self.raza}")
26         super().__del__()  # Apelam destructorul clasei de baza
27
28 def main():
29     c = Cerc(1.1, 2.2, 10)
30     c.afisare()
31
32 if __name__ == "__main__":
33     main()
```

La rularea exemplului 5.13 efectul afisat în consola este urmatorul:

```
Constructor punct: x=1.1 y=2.2

Constructor cerc: raza=10
```

```
Cerc:
centru: x=1.1   y=2.2
raza: 10

Destructor cerc: raza=10

Destructor punct: x=1.1 y=2.2
```

În clasa `Punct`, metoda `afisare()` afișează coordonatele x și y, respectiv, în clasa `Cerc`, metoda `afisare()` este redefinită pentru a adăuga informații suplimentare, respectiv valoarea raza, și pentru a include și informațiile de la clasa de bază. Atunci când se apelează `c.afisare()`, metoda `afisare()` din clasa `Cerc` este utilizată, deoarece aceasta supradefinește metoda `afisare()` din `Punct`. Constructorii și destructoarele din ambele clase sunt apelate în ordinea moștenirii, cu constructorul clasei de bază apelat înainte de constructorul clasei derivate și destructorul clasei derivate apelat înainte de destructorul clasei de bază.

5.2.5 Supradefinirea operatorilor în clasele derivate

Funcţiile operator membre ale clasei de bază sunt moştenite de clasele derivate şi, dacă este necesar, pot fi supradefinite în clasele derivate, ca toate celelalte funcţii membre. O excepţie o constituie operatorul de atribuire, care se supune următoarelor reguli:

- dacă operatorul este supradefinit în clasa derivată, funcţia `operator=` preia sarcina efectuării atribuirilor pentru toţi membrii, şi pentru cei ai clasei de bază, chiar dacă este supradefinit operatorul în clasa de bază;
- dacă nu este supradefinită funcţia `operator=` în clasa derivată, dar este definită în cea de bază, atribuirea pentru membrii clasei de bază se face apelându-se funcţia `operator=`, pentru ceilalţi membri făcându-se atribuirea membru cu membru;
- dacă nu este supradefinit operatorul de atribuire în nici una din clase, atribuirea se face membru cu membru.

În Exemplu 5.14 se declară clasa `Segment` care defineşte un segment de dreaptă prin coordonatele extremităţilor sale şi clasa derivată din aceasta, `Dreptunghi`, care preia segmentul de dreaptă ca şi diagonală a sa.

Exemplu 5.14 Exemplu, în limbaj C++, pentru declarația unui `segment` de dreaptă și derivarea acestuia în clasa `dreptunghi` pentru a prelua segmentul de dreaptă ca si diagonala a sa.

```
1 #include <iostream>
2
3 class segment
4 {
5 protected: // se permite accesul datelor membre din clasele
    ↪ derivate
6       int x1, y1, x2, y2;
```

```
7  public:
8        segment(int a1=0, int b1=0, int a2=0, int b2=0)
9          {
10              std::cout<<"\nConstructor segment";
11              if (a1<a2)
12              { x1=a1; x2=a2; }
13              else
14              { x1=a2; x2=a1; }
15              if (b1<b2)
16              { y1=b1; y2=b2; }
17              else
18              { y1=b2; y2=b1; }
19         }
20
21     segment operator=(segment s)
22         {
23              std::cout<<"\nApel operator=() pentru segment";
24              x1=s.x1; y1=s.y1; x2=s.x2; y2=s.y2;
25              return *this;
26         }
27
28     segment operator+(segment s)
29         {
30              std::cout<<"\nApel operator+() pentru segment";
31              segment aux;
32              aux.x1=x1+s.x1;
33              aux.y1=y1+s.y1;
34              aux.x2=x2+s.x2;
35              aux.y2=y2+s.y2;
36              return aux;
37         }
38
39     void afisare()
40         {
41              std::cout<<"\nx1="<<x1<<" y1="<<y1<<" x2="<<x2<<" y2="
                    ↪ <<y2;
42         }
43 };
44
45 class dreptunghi: public segment
46 {
```

```
47 public:
48         dreptunghi(segment s):segment(s)
49         {
50             std::cout<<"\nConstructor - conversie segment->
                ↪ dreptunghi";
51         }
52
53     dreptunghi(int a1=0, int b1=0, int a2=0, int b2=0):segment(a1,
          ↪ b1,a2,b2)
54         {
55             std::cout<<"\nConstructor dreptunghi";
56         }
57
58     long aria()
59         {
60             return (x2-x1)*(y2-y1);
61         }
62
63     void afisare()
64         {
65             segment::afisare(); // se apeleaza functia afisare() a
                ↪  clasei de baza
66             std::cout<<"\naria="<<aria();
67         }
68 };
69
70 int main()
71 {
72   dreptunghi d1(1, 1, 3, 3), d2(5, 5, 8, 8), d3;
73   d3=d1;
74   d1.afisare();
75   d3=d1+d2;
76   d3.afisare();
77   return 0;
78 }
```

La rularea exemplului 5.14 efectul afișat în consolă este următorul:

```
Constructor segment  // pentru fiecare din cele 3 obiecte dreptunghi se
Constructor dreptunghi  // apeleaza constructorul segment si apoi
Constructor segment  // constructorul dreptunghi
Constructor dreptunghi
```

```
Constructor segment
Constructor dreptunghi

Apel operator=() pentru segment // Pentru atribuirea d3=d1, in absenta
// supradefinirii operator=() pentru clasa dreptunghi, compilatorul apeleaza
// operator=() definit pentru segment

x1=1 y1=1 x2=3 y2=3
aria=4
Apel operator+() pentru segment  // Extinderea functionarii operator+()
// definit pentru segment asupra obiectelor dreptunghi

Constructor segment // Rezultatul returnat de operator+() e preluat  de
// un obiect temporar segment

Constructor  -  conversie segment - dreptunghi // Conversie necesara atribuirii

Apel operator=() pentru segment // Apel similar celui anterior al
// functiei operator=()
x1=6 y1=6 x2=11 y2=11
aria=25
```

În Python, supradefinirea se face pur și simplu prin definirea unei metode cu același nume în clasa derivată. De asemenea, pentru a apela metoda din clasa de bază, se poate folosi super().

Folosind acelasi exemplu cu clasele segment și dreptunghi, în exemplul următor, 5.15, se regăsește conceptul de supradefinire în limbajul Python.

Exemplu 5.15 Exemplu, în limbajul Python, pentru declarația unui **segment** de dreapta și derivarea acestuia în clasa **dreptunghi** pentru a prelua segmentul de dreapta ca și diagonală a sa.

```
1  class Segment:
2      def __init__(self, x1=0, y1=0, x2=0, y2=0):
3          self.x1 = min(x1, x2)
4          self.y1 = min(y1, y2)
5          self.x2 = max(x1, x2)
6          self.y2 = max(y1, y2)
7          print("\nConstructor segment")
8
9      def __add__(self, other):
10         if isinstance(other, Segment):
11             print("\nApel operator+() pentru segment")
12             return Segment(
```

```
13                  self.x1 + other.x1,
14                  self.y1 + other.y1,
15                  self.x2 + other.x2,
16                  self.y2 + other.y2
17              )
18          return NotImplemented
19
20      def __eq__(self, other):
21          if isinstance(other, Segment):
22              print("\nApel operator=() pentru segment")
23              self.x1, self.y1, self.x2, self.y2 = other.x1, other.
                    ↪ y1, other.x2, other.y2
24          return self
25
26      def display(self):
27          print(f"\nx1={self.x1} y1={self.y1} x2={self.x2} y2={self.
                ↪ y2}")
28
29
30  class Dreptunghi(Segment):
31      def __init__(self, x1=0, y1=0, x2=0, y2=0):
32          super().__init__(x1, y1, x2, y2)
33          print("\nConstructor dreptunghi")
34
35      def area(self):
36          return (self.x2 - self.x1) * (self.y2 - self.y1)
37
38      def display(self):
39          super().display()
40          print(f"\naria={self.area()}")
41
42
43  # Exemplu de utilizare
44  if __name__ == "__main__":
45      d1 = Dreptunghi(1, 1, 3, 3)
46      d2 = Dreptunghi(5, 5, 8, 8)
47      d3 = Dreptunghi()
48
49      d3 = d1
50      d1.display()
51      d3 = d1 + d2
```

```
52      d3.display()
```

La rularea exemplului 5.15 efectul afișat în consolă este următorul:

```
Constructor segment
Constructor dreptunghi
Constructor segment
Constructor dreptunghi
Constructor segment
Constructor dreptunghi

x1=1 y1=1 x2=3 y2=3
aria=4
Apel operator+() pentru segment
Constructor segment
x1=6 y1=6 x2=11 y2=11
```

În acest exemplu, constructorii sunt definiți cu `__init__`, respectiv constructorul din clasa de bază este apelat folosind `super().__init__()`. Supradefinirea operatorilor se face prin metode speciale, cum ar fi `__add__` pentru operatorul `+` și `__eq__` pentru operatorul `=` (atribuire). Metodele din clasa derivată pot apela metodele din clasa de bază folosind `super().metoda()`.

5.3 Ierarhii de clase

Clasele derivate pot fi folosite ca și clase de bază pentru a crea noi clase derivate, formând astfel ierarhii de clase. Procesul începe cu o clasă relativ simplă și, pe măsură ce se adaugă noi clase derivate, fiecare derivare introduce noi proprietăți, iar clasele devin tot mai complexe. Fiecare clasă derivată moștenește proprietățile tuturor claselor anterioare utilizate succesiv ca și clase de bază.

În exemplul 5.16 se creează o ierarhie clasa `poz` -> clasa `punct` -> clasa `caracter`. Clasa `caracter` moşteneşte proprietăţile celorlalte două.

Clasa `poz` reprezintă poziţia punctelor în plan prin coordonate carteziene. Clasa `punct` reprezintă puncte în plan, prin coordonate carteziene. Pentru afişare, punctele au precizate proprietăţile `vizibil` (1 permite afişarea, 0 nu) şi `culoarea` folosită la afişare; marcarea punctului se face prin caracterul '*'. Clasa `caracter`, pe lângă `coordonate`, respectiv proprietăţile de `culoare` şi `vizibil`, permite precizarea caracterului `c` ce se va afişa.

Exemplu 5.16 Exemplu, în limbaj C++, de ierarhii de clase

```
1 #include <iostream>
2 #include <conio.h>
3
4 class poz                    // declaratia clasei poz
5 {
```

```
6  protected:
7      int x, y;
8  public:
9      poz(int abs=0, int ord=0)
10       {
11          std::cout<<"\nconstructor poz: ";
12          x=abs; y=ord;
13          std::cout<<"x="<<x<<"\ty="<<y ;
14       }
15
16     ~poz()
17       {
18         std::cout<<"\ndestructor poz:";
19         std::cout<<"x="<<x<<"\ty="<<y ;
20       }
21
22     void af()
23       {
24         std::cout<<"\npoz: x="<<x<<" y="<<y;
25       }
26
27     void depl(int dx, int dy)
28       {
29         x+=dx;
30         y+=dy;
31       }
32 };
33
34 class punct:public poz // declaratia clasei punct, derivata din
     ↪ clasa poz
35 {
36 protected:
37     int vizibil;      //vizibil=0-punct invizibil
38     int culoare;
39 public:
40     punct(int abs, int  ord, int cul) : poz(abs, ord)
41       {
42         std::cout<<"\nconstructor punct: ";
43         vizibil=1;
44         culoare=cul;
45         std::cout<<" vizibil="<<vizibil<<"\tculoare="<<cul;
```

```
46        }
47
48      ~punct()
49        {
50          std::cout<<"\ndestructor punct";
51          std::cout<<"vizibil="<<vizibil<<"\tculoare="<<culoare;
52        }
53
54      void cul(int c)
55        {
56          culoare=c;
57        }
58
59      void viz(int v)
60        {
61          vizibil=v;
62        }
63
64      void afisare()
65        {
66          if (vizibil)
67          {
68            // afiseaza
69          }
70        }
71  };
72
73  class caracter:public punct // declaratia clasei caracter,
      ↪ derivata din clasa punct; ea va mosteni atat membrii clasei
      ↪ punct, cat si ai clasei poz
74  {
75      char c; // caracterul specific
76  public:
77      caracter(int abs, int ord, int cul, char ch) : punct(abs, ord,
          ↪ cul)
78        {
79          std::cout<<"\nconstructor caracter: ";
80          c=ch;
81          std::cout<<"caracter="<<c;
82        }
83
```

```
84      ~caracter()
85        {
86          std::cout<<"\ndestructor caracter: ";
87          std::cout<<"caracter="<<c;
88        }
89
90      void afisare()
91        {
92          if (vizibil)
93          {
94            // afiseaza
95          }
96        }
97  };
98
99  int main()
100 {
101   poz p1(5, 5);
102   p1.af();
103   getch();
104
105   punct pct1(7, 7, 7);
106   pct1.af();
107   pct1.afisare();
108   pct1.cul(15);
109   pct1.depl(1, 1);
110   pct1.afisare();
111   getch();
112
113   caracter c1(12, 12, 15, '#');
114   c1.afisare();
115   getch();
116
117   return 0;
118 }
```

La rularea exemplului 5.16 efectul afisat în consola este urmatorul:

```
constructor poz:  x=5 y=5 // se creeaza obiectul p1
poz:   x=5 y=5
constructor poz:  x=7 y=7 // se creeaza obiectul pct1, apelandu-se
// intai constructorul poz() si apoi punct ()
```

```
constructor punct: vizibil=1  culoare=7
poz: x=7 y=7
 *
     *
constructor poz:  x=12 y=12   // se creeaza obiectul c1, apelandu-se
//  intai constructorul poz(), apoi punct (),

constructor punct: vizibil=1  culoare=15  // apoi caracter()
constructor caracter: caracter = #
  #
destructor caracter: caracter = #  // se distruge obiectul c1,
// apelandu-se destructor punct: vizibil=1  culoare=15
// intai destructorul ~poz(), apoi destructor poz:  x=12 y=12
// ~punct (), apoi ~caracter()

destructor punct: vizibil=1  culoare=15 // se distruge obiectul pct1, apelandu-se
destructor poz:  x=8 y=8 // intai destructorul ~poz(), apoi ~punct ()

destructor poz:  x=5 y=5 // se distruge obiectul p1, apelandu-se
// destructorul ~poz()
```

Clasa `caracter` derivă din clasa `punct`, care la rândul său este derivată din clasa `poz`. Astfel, ea moștenește membrii `x` și `y` de la `poz`, `vizibil` și `culoare` de la `punct` și adaugă membrul `caracter`. Funcția `af()` din clasa `poz` este moștenită și poate fi apelată pentru un obiect `caracter`. Pe de altă parte, funcția `afisare()` definită în clasa `punct` este suprascrisă în clasa `caracter`, iar pentru un obiect de tip `caracter`, va fi apelată funcția definită în această ultimă clasă.

Prin mesajele afişate, se poate vedea că ordinea apelării constructorilor la crearea unui obiect de tip `caracter` este, în ordine, constructor `poz`, constructor `punct`, constructor `caracter`. Destructorii sunt apelaţi în ordine inversă.

În Python, ierarhiile de clase sunt structuri de moștenire, similare cu cele din C++, care permit crearea de clase derivate din clase de bază, extinzând și personalizând comportamentul acestora. Plecând de la exemplul 5.16, în continuare este descris conceptul de ierarhie de clase folosind limbajul Python.

Exemplu 5.17 Exemplu, în limbaj Python, de ierarhii de clase

```
1 class Poz:
2     def __init__(self, x=0, y=0):
3         print(f"constructor poz: x={x} y={y}")
4         self.x = x
5         self.y = y
```

```
6
7       def __del__(self):
8           print(f"destructor poz: x={self.x} y={self.y}")
9
10      def af(self):
11          print(f"poz: x={self.x} y={self.y}")
12
13      def depl(self, dx, dy):
14          self.x += dx
15          self.y += dy
16
17
18  class Punct(Poz):
19      def __init__(self, x, y, culoare):
20          super().__init__(x, y)
21          print(f"constructor punct: vizibil=1 culoare={culoare}")
22          self.vizibil = 1
23          self.culoare = culoare
24
25      def __del__(self):
26          print(f"destructor punct: vizibil={self.vizibil} culoare={
                ↪ self.culoare}")
27          super().__del__()
28
29      def cul(self, culoare):
30          self.culoare = culoare
31
32      def viz(self, vizibil):
33          self.vizibil = vizibil
34
35      def afisare(self):
36          if self.vizibil:
37              print(f"poz: x={self.x} y={self.y} *")
38
39
40  class Caracter(Punct):
41      def __init__(self, x, y, culoare, caracter):
42          super().__init__(x, y, culoare)
43          print(f"constructor caracter: caracter={caracter}")
44          self.caracter = caracter
45
```

```
46      def __del__(self):
47          print(f"destructor caracter: caracter={self.caracter}")
48          super().__del__()
49
50      def afisare(self):
51          if self.vizibil:
52              print(f"poz: x={self.x} y={self.y} {self.caracter}")
53
54
55  # Exemplu de utilizare
56  if __name__ == "__main__":
57      p1 = Poz(5, 5)
58      p1.af()
59      print()
60
61      pct1 = Punct(7, 7, 7)
62      pct1.af()
63      pct1.afisare()
64      pct1.cul(15)
65      pct1.depl(1, 1)
66      pct1.afisare()
67      print()
68
69      c1 = Caracter(12, 12, 15, '#')
70      c1.afisare()
71      print()
```

La rularea exemplului 5.17 efectul afisat în consola este urmatorul:

```
constructor poz: x=5 y=5
poz: x=5 y=5

constructor poz: x=7 y=7
constructor punct: vizibil=1 culoare=7
poz: x=7 y=7
poz: x=7 y=7 *
poz: x=8 y=8 *

constructor poz: x=12 y=12
constructor punct: vizibil=1 culoare=15
constructor caracter: caracter=#
poz: x=12 y=12 #
```

```
destructor poz: x=5 y=5
destructor punct: vizibil=1 culoare=15
destructor poz: x=8 y=8
destructor caracter: caracter=#
destructor punct: vizibil=1 culoare=15
destructor poz: x=12 y=12
```

Clasa `Poz` este clasa de bază. Aceasta definește coordonatele x și y și include metode pentru a afișa și deplasa punctul. Clasa `Punct` moștenește `Poz`. Adaugă noi atribute pentru vizibilitate și culoare, și include metoda afisare care folosește aceste atribute. Clasa `Caracter` moștenește `Punct`. Adaugă un caracter suplimentar și redefinește metoda afisare pentru a afișa acest caracter.

5.4 Moştenirea multiplă

Conceptul de moștenire permite generarea unor clase noi care pot prelua caracteristici din mai multe clase de bază. Moștenirea multiplă oferă o flexibilitate sporită în definirea claselor, rezultând structuri de clase mai complexe.

Sintaxa pentru definirea unei clase derivate `D` este următoarea:

```
1 class D: specif_acces clasa_baza1, specif_acces clasa_baza2, ...
```

În lista de clase de bază, pentru fiecare clasă menționată, se indică specificatorul de acces corespunzător. Principiile aplicabile în cazul moștenirii simple și al ierarhiilor simple de clase se aplică și în contextul moștenirii multiple.

În exemplul 5.18 se declară ierarhia de clase:

```
1 poz -> punct      |
2                   |  -> strpoz
3 string            |
```

Clasele `Poz` şi `Punct` au semnificaţia descrisă în exemplul 5.18. Se declară clasa `String` pentru reprezentarea de şiruri de caractere. Clasa `Strpoz` este construită prin derivare multiplă, având ca şi clase de bază clasele `Punct` şi `String`. La afişarea unui obiect de tip `Strpoz`, se afişează şirul de caracter, `str`, la poziţia indicată prin coordonatele `x` şi `y`, cu precizarea culorii de afişare, prin membrul `culoare`, în funcţie de proprietatea `vizibil`.

Exemplu 5.18 Exemplu, în limbaj C++, de moștenire multiplă.

```
1 #include <iostream>
2 #include <conio.h>
3 #include <stdio.h>
4 #include <cstring>
5
6 class poz; // Forward declaration
```

```
7
8  class punct; // Forward declaration
9
10 class poz {
11 protected:
12     int x, y;
13 public:
14     poz(int abs = 0, int ord = 0); // Constructor cu parametri
       ↪ impliciti
15     poz(const poz &p); // Constructor de copiere
16     ~poz();
17     void af();
18     void depl(int dx, int dy); // Modificarea coordonatelor
19 };
20
21 class punct : public poz {
22 protected:
23     int vizibil; // 0 - invizibil, 1 - vizibil
24     int culoare; // Culoarea de afisare
25 public:
26     punct(int abs = 0, int ord = 0, int cul = 7);
27     punct(const punct &);
28     ~punct();
29     void cul(int);
30     void viz(int);
31     void afisare();
32
33     // Operator de conversie poz -> punct mutat aici
34     punct(poz p) : poz(p), vizibil(1), culoare(7) {}
35 };
36
37 class string {
38 protected:
39     int ncar; // Lungimea sirului
40     char *str;
41 public:
42     string(int n = 0);
43     string(const char *s);
44     string(const string& s); // Constructor de copiere
45     ~string();
46     void afisare();
```

```
47 };
48
49 class strpoz : public punct, public string {
50 public:
51     strpoz(int, int, int, const char *);
52     ~strpoz();
53     void afisare();
54 };
55
56 // Implementari
57 poz::poz(int abs, int ord) {
58     std::cout << "\n\nConstructor poz " << this;
59     std::cout << "\n&x=" << &x << "  &y=" << &y;
60     x = abs;
61     y = ord;
62 }
63
64 poz::poz(const poz &p) {
65     std::cout << "\n\nConstructor copiere poz";
66     x = p.x;
67     y = p.y;
68 }
69
70 poz::~poz() {
71     std::cout << "\n\nDestructor poz.";
72 }
73
74 void poz::af() {
75     std::cout << "\n\npoz: x=" << x << " y=" << y;
76 }
77
78 void poz::depl(int dx, int dy) {
79     x += dx;
80     y += dy;
81 }
82
83 punct::punct(int abs, int ord, int cul) : poz(abs, ord) {
84     std::cout << "\nConstructor punct:" << this;
85     std::cout << "\n&vizibil=" << &vizibil;
86     std::cout << "  &culoare=" << &culoare;
87     vizibil = 1;
```

```
88      culoare = cul;
89  }
90
91  punct::punct(const punct &p) : poz(p) {
92      std::cout << "\nConstructor copiere punct:" << this;
93      vizibil = p.vizibil;
94      culoare = p.culoare;
95  }
96
97  punct::~punct() {
98      std::cout << "\nDestructor punct";
99  }
100
101 void punct::cul(int c) {
102     culoare = c;
103 }
104
105 void punct::viz(int v) {
106     vizibil = v;
107 }
108
109 void punct::afisare() {
110     if (vizibil) {
111         std::cout << "\nAfisare punct:";
112         std::cout << "\nCuloare text: " << culoare;
113         std::cout << "\nCoordonate: " << x << " , " << y;
114     }
115 }
116
117 string::string(int n) {
118     ncar = n;
119     str = new char[ncar + 1];
120     str[0] = '\0';
121     std::cout << "Constructor 1 - string";
122 }
123
124 string::string(const char *s) {
125     ncar = strlen(s);
126     str = new char[ncar + 1];
127     strcpy(str, s);
128     std::cout << "\nConstructor 2 - string";
```

```
129 }
130
131 string::string(const string &s) {
132     ncar = s.ncar;
133     str = new char[ncar + 1];
134     strcpy(str, s.str);
135     std::cout << "\nConstructor copiere - string";
136 }
137
138 string::~string() {
139     delete[] str;
140     std::cout << "\n\nDestructor - string";
141 }
142
143 void string::afisare() {
144     std::cout << "\n\nString: " << str << "\n";
145 }
146
147 strpoz::strpoz(int abs, int ord, int cul, const char *s) : punct(
    ↪ abs, ord, cul), string(s) {
148     std::cout << "\n\nConstructor - strpoz";
149 }
150
151 strpoz::~strpoz() {
152     std::cout << "\n\nDestructor - strpoz";
153 }
154
155 void strpoz::afisare() {
156     std::cout << "\nAfisare strpoz:\n";
157     std::cout << "\nCuloare text: " << culoare;
158     std::cout << "\nCoordonate: " << x << ", " << y;
159     std::cout << "\nSirul: " << str;
160 }
161
162 int main() {
163     poz p1(5, 5), p2(510);
164     p1.af();
165     p2.af();
166
167     punct pct1(10, 12, 7);
168     pct1.af();
```

```
169     pct1.afisare();
170     pct1.cul(15);
171     pct1.depl(3, 3);
172     pct1.afisare();
173
174     string s1("exemplu");
175     s1.afisare();
176
177     strpoz str1(22, 22, 6, "aaa");
178     str1.afisare();
179
180     return 0;
181 }
```

La rularea exemplului 5.18 efectul afisat în consola este urmatorul:

```
Constructor poz 0x7ffdbd45ad00 &x=0x7ffdbd45ad00  &y=0x7ffdbd45ad04
Constructor poz 0x7ffdbd45ad08 &x=0x7ffdbd45ad08  &y=0x7ffdbd45ad0c
poz: x=5 y=5
poz: x=510 y=0
Constructor poz 0x7ffdbd45ad10 &x=0x7ffdbd45ad10  &y=0x7ffdbd45ad14
Constructor punct:0x7ffdbd45ad10 &vizibil=0x7ffdbd45ad18  &culoare=0x7ffdbd45ad1c
poz: x=10 y=12
Afisare punct:
Culoare text: 7
Coordonate: 10 , 12
Afisare punct:
Culoare text: 15
Coordonate: 13 , 15
Constructor 2 - string
String: exemplu
Constructor poz 0x7ffdbd45ad30 &x=0x7ffdbd45ad30  &y=0x7ffdbd45ad34
Constructor punct:0x7ffdbd45ad30 &vizibil=0x7ffdbd45ad38  &culoare=0x7ffdbd45ad3c
Constructor 2 - string
Constructor - strpoz
Afisare strpoz:
Culoare text: 6
Coordonate: 22, 22
Sirul: aaa
Destructor - strpoz
Destructor - string
Destructor punct
```

```
Destructor poz.
Destructor - string
Destructor punct
Destructor poz.
Destructor poz.
Destructor poz.
```

În definiția clasei `strpoz`, se indică faptul că aceasta derivă public din clasele `pozitie` și `string`.

Fiecare constructor al clasei `strpoz` are rolul de a transfera datele către constructorii claselor de bază. Constructorii și destructorii claselor de bază sunt invocați automat atunci când se creează, respectiv distrug obiecte derivate. Ordinea în care sunt apelați constructorii depinde de secvența listată a claselor de bază în definiția clasei derivate, iar constructorul clasei derivate este invocat ultimul. Destructorii sunt apelați în ordine inversă.

Clasa derivată conține toți membrii claselor de bază, dar poate accesa doar pe acelea declarate cu acces public sau protected. Clasele derivate au posibilitatea de a suprascrie funcțiile existente în clasele de bază. Membrii și funcțiile cu același nume pot fi accesate utilizând operatorul de rezoluție. De exemplu, pentru clasa `strpoz`, funcția membră de afișare poate fi definită astfel:

```
1 void strpoz::afisare()
2 {
3   cout<<"\\n Strpoz:";
4   pozitie::afisare();
5   string::afisare();
6   cout<<"\\n Culoare:"<<c<<endl;
7 }
```

iar, în funcţia main() se poate include linia de program:

```
1 sp3.pozitie::afisare();
```

În Python, conceptul de moștenire multiplă este similar cu cel din C++, oferind flexibilitate în proiectarea claselor complexe. Clasele noi pot moșteni caracteristici de la mai multe clase de bază. Sintaxa folosită pentru declararea unei clase derivate `D` este:

```
1 class D(clasa_baza1, clasa_baza2, ...):
2     # corpul clasei
```

Ordinea în care sunt moștenite clasele de bază afectează ordinea în care constructorii și metodele acestora sunt apelate. Toate clasele de bază contribuie cu atribute și metode la clasa derivată, însă membrii care au același nume pot fi suprascrisi sau pot fi accesați explicit folosind numele clasei. În Python, moștenirea multiplă este reglementată de metoda MRO (Method Resolution Order), care determină ordinea în care sunt apelate metodele din ierarhia de clase.

În exemplul următor se declară o ierarhie de clase:

```
class Pozitie:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def afisare(self):
        print(f"Coordonate: ({self.x}, {self.y})")

class String:
    def __init__(self, s=""):
        self.s = s

    def afisare(self):
        print(f"Sir: {self.s}")

class StrPoz(Pozitie, String):
    def __init__(self, x, y, s):
        Pozitie.__init__(self, x, y)
        String.__init__(self, s)

    def afisare(self):
        Pozitie.afisare(self)
        String.afisare(self)
```

Clasa `StrPoz` este derivată din clasele `Pozitie` și `String`. La afișarea unui obiect de tip `StrPoz`, se afișează coordonatele și șirul de caractere moștenite din clasele de bază:

```
sp = StrPoz(22, 22, "Exemplu")
sp.afisare()
```

Plecând de la exemplul 5.18, moștenirea multiplă echivalentă în Python este următoarea:

Exemplu 5.19 Exemplu, în Python, de moștenire multiplă.

```
import copy

class Pozitie:
    def __init__(self, abs=0, ord=0):
        self.x = abs
        self.y = ord
        print(f"\n\nConstructor Pozitie {self}")
        print(f"&x={id(self.x)} &y={id(self.y)}")

```

```
10      # Copy constructor equivalent
11      def copy(self):
12          print("\n\nCopy Constructor Pozitie")
13          return copy.deepcopy(self)
14
15      def __del__(self):
16          print("\n\nDestructor Pozitie")
17
18      def afisare(self):
19          print(f"\n\nPoz: x={self.x} y={self.y}")
20
21      def depl(self, dx, dy):
22          self.x += dx
23          self.y += dy
24
25
26  class Punct(Pozitie):
27      def __init__(self, abs=0, ord=0, cul=7):
28          super().__init__(abs, ord)
29          self.vizibil = 1
30          self.culoare = cul
31          print(f"\nConstructor Punct: {self}")
32          print(f"&vizibil={id(self.vizibil)} &culoare={id(self.
                ↪ culoare)}")
33
34      # Copy constructor equivalent
35      def copy(self):
36          print("\nCopy Constructor Punct:")
37          return copy.deepcopy(self)
38
39      def __del__(self):
40          print("\nDestructor Punct")
41
42      def cul(self, c):
43          self.culoare = c
44
45      def viz(self, v):
46          self.vizibil = v
47
48      def afisare(self):
49          if self.vizibil:
```

```
50              print("\nAfisare Punct:")
51              print(f"culoare text: {self.culoare}")
52              print(f"coordonate: {self.x}, {self.y}")
53
54      # Operator conversion equivalent
55      @staticmethod
56      def from_pozitie(poz):
57          print("\nConverting Pozitie to Punct...")
58          pct = Punct(poz.x, poz.y, 7)
59          return pct
60
61
62  class Sir:
63      def __init__(self, s=""):
64          self.ncar = len(s)
65          self.str = s
66          print("\nConstructor String")
67
68      def __del__(self):
69          print("\n\nDestructor String")
70
71      def afisare(self):
72          print(f"\n\nString: {self.str}\n")
73
74
75  class StrPoz(Punct, Sir):
76      def __init__(self, abs, ord, cul, s):
77          Punct.__init__(self, abs, ord, cul)
78          Sir.__init__(self, s)
79          print("\n\nConstructor StrPoz")
80
81      def __del__(self):
82          print("\n\nDestructor StrPoz")
83
84      def afisare(self):
85          print("\nAfisare StrPoz:\n")
86          print(f"culoare text: {self.culoare}")
87          print(f"coordonate: {self.x}, {self.y}")
88          print(f"sirul: {self.str}")
89
90
```

```
91 # Testarea claselor
92 if __name__ == "__main__":
93     p1 = Pozitie(5, 5)
94     p2 = Pozitie(510)
95     p1.afisare()
96     p2.afisare()
97
98     # Copy constructor usage
99     p3 = p1.copy()
100    p3.afisare()
101
102    pct1 = Punct(10, 12, 7)
103    pct1.afisare()
104
105    # Conversion operator equivalent usage
106    pct2 = Punct.from_pozitie(p1)
107    pct2.afisare()
108
109    pct1.cul(15)
110    pct1.depl(3, 3)
111    pct1.afisare()
112
113    s1 = Sir("exemplu")
114    s1.afisare()
115
116    str1 = StrPoz(22, 22, 6, "aaa")
117    str1.afisare()
```

La rularea exemplului 5.19 efectul afisat în consola este urmatorul:

```
Constructor Pozitie  &x=11758120 &y=11758120
Constructor Pozitie  &x=134510714279984 &y=11757960
Poz: x=5 y=5
Poz: x=510 y=0
Copy Constructor Pozitie
Poz: x=5 y=5
Constructor Pozitie &x=11758280 &y=11758344
Constructor Punct:  &vizibil=11757992 &culoare=11758184
Afisare Punct:
culoare text: 7
coordonate: 10, 12
Converting Pozitie to Punct...
```

```
Constructor Pozitie &x=11758120 &y=11758120
Constructor Punct:  &vizibil=11757992 &culoare=11758184
Afisare Punct:
culoare text: 7
coordonate: 5, 5
Afisare Punct:
culoare text: 15
coordonate: 13, 15
Constructor String
String: exemplu
Constructor Pozitie &x=11758664 &y=11758664
Constructor Punct:  &vizibil=11757992 &culoare=11758152
Constructor String
Constructor StrPoz
Afisare StrPoz:
culoare text: 6
coordonate: 22, 22
sirul: aaa
Destructor Pozitie
Destructor Pozitie
Destructor Pozitie
Destructor Punct
Destructor Punct
Destructor String
Destructor StrPoz
```

În exemplul 5.19, `Pozitie` este clasă de bază în Python. Constructorul inițializează coordonatele x și y, iar metoda `depl` modifică aceste coordonate. Metoda numită `afisare` le tipărește. Clasa derivată `Punct` moștenește de la `Pozitie`. Aceasta are membri suplimentari pentru vizibilitate și culoare. Metodele `viz` și `cul` modifică acești membri, iar metoda `afisare` îi afișează. Clasa `Sir`: este o clasă simplă care gestionează un șir de caractere. Clasa `StrPoz` moștenește multiplu din `Punct` și `Sir`. Constructorul inițializează ambii membri din clasele de bază, iar metoda `afisare` afișează toate informațiile.

6. Conceptul de polimorfism

Polimorfism în limbaje statice (C++) versus limbaje dinamice (Python)
Tipuri de polimorfism
Supradefinirea (eng. overriding) și supraîncarcarea (eng. overloading) în OOP
Funcții virtuale
Funcții virtuale pure. Clase abstracte

Polimorfismul este un concept fundamental în programarea orientată pe obiecte (OOP), care se referă la abilitatea obiectelor de a adopta mai multe forme sau comportamente. Cuvântul `polimorfism` provine din greacă, unde `poly` înseamnă `multe`, iar `morph` înseamnă `forme`. În esență, polimorfismul permite unui obiect să se comporte diferit în funcție de contextul în care este utilizat, permițând crearea unor soluții mai flexibile și mai scalabile în cod. În C++, polimorfismul este bine definit și controlat, oferind siguranță și predictibilitate la nivelul tipurilor și funcțiilor, dar poate implica un nivel de complexitate suplimentară. În Python, polimorfismul este mai natural și flexibil datorită tipurilor dinamice și principiului duck typing (un obiect este considerat de un anumit tip dacă are toate metodele și proprietățile necesare tipului respectiv), dar poate duce la erori mai greu de detectat dacă tipurile obiectelor nu sunt gestionate cu atenție. Indiferent de limbajul folosit, polimorfismul contribuie la scrierea unui cod care este modular și mai ușor de întreținut, încurajând reutilizarea codului și extensibilitatea aplicațiilor.

Mai specific, polimorfismul oferă posibilitatea de a folosi o interfață comună sau o metodă pentru obiecte de tipuri diferite, unde comportamentul acelei metode este adaptat în funcție de tipul specific al obiectului care o implementează. Acest lucru înseamnă că, într-un program orientat pe obiecte, se pot defini metode generice care pot lucra cu orice obiect de un tip derivat dintr-o clasă comună, fără a fi nevoie să cunoaștem tipul exact al obiectului la momentul scrierii codului. Astfel, un comportament specific poate fi determinat dinamic în funcție de tipul real al obiectului la runtime.

Polimorfismul se bazează adesea pe moștenire pentru a funcționa eficient. De exemplu, o clasă de bază poate defini o metodă abstractă sau virtuală, iar clasele derivate pot oferi implementări diferite pentru această metodă. Programatorii pot apoi apela metoda respectivă

folosind un pointer de tipul clasei de bază, dar comportamentul specific este determinat de implementarea din clasa derivată (care este selectată în funcție de tipul efectiv al obiectului).

În exemplul 6.1, vom folosi clasele Animal, Caine și Pisica pentru a demonstra polimorfismul în C++.

Exemplu 6.1 Exemplu, în limbaj C++, de polimorfism.

```cpp
#include <iostream>
using namespace std;

// Clasa de baza Animal
class Animal {
public:
    virtual void vorbeste() {
        cout << "Animalul face un sunet." << endl;
    }
};

// Clasa derivata Caine
class Caine : public Animal {
public:
    void vorbeste() override {
        cout << "Cainele latra." << endl;
    }
};

// Clasa derivata Pisica
class Pisica : public Animal {
public:
    void vorbeste() override {
        cout << "Pisica miauna." << endl;
    }
};

int main() {
    Animal* animale[] = {new Caine(), new Pisica(), new Animal()};

    // Polimorfism: Metoda "vorbeste()" se comporta diferit in
    ↪ functie de tipul obiectului.
    for (int i = 0; i < 3; i++) {
        animale[i]->vorbeste();
    }
```

```
35
36      // Eliberare memorie
37      for (int i = 0; i < 3; i++) {
38          delete animale[i];
39      }
40
41      return 0;
42  }
```

La rularea exemplului 6.1 efectul afisat în consola este următorul:

```
Cainele latra.
Pisica miauna.
Animalul face un sunet.
```

În exemplul 6.1 `Animal` este o clasă de bază care definește o metodă virtuală numită `vorbeste()`. Clasele `Caine` și `Pisica` moștenesc clasa `Animal` și suprascriu metoda `vorbeste()`. În funcția `main()`, folosim un array de pointeri la `Animal`, dar fiecare obiect referit prin elementele array-ului este de un tip derivat diferit. La runtime, metoda potrivită este apelată în funcție de tipul efectiv al obiectului (`Caine` sau `Pisica`), demonstrând polimorfismul.

În Python, vom folosi un exemplu similar, dar datorită dinamicității tipurilor și duck typing-ului, implementarea va fi mai simplă.

Exemplu 6.2 Exemplu, în Python, de polimorfism.

```
1  # Clasa de baza Animal
2  class Animal:
3      def vorbeste(self):
4          print("Animalul face un sunet.")
5
6  # Clasa derivata Caine
7  class Caine(Animal):
8      def vorbeste(self):
9          print("Cainele latra.")
10
11 # Clasa derivata Pisica
12 class Pisica(Animal):
13     def vorbeste(self):
14         print("Pisica miauna.")
15
16 # Polimorfism: Metoda "vorbeste()" se comporta diferit in functie
       ↪ de tipul obiectului
17 animale = [Caine(), Pisica(), Animal()]
18
```

```
19 for animal in animale:
20     animal.vorbeste()
```

La rularea exemplului 6.2 efectul afisat în consola este următorul:

```
Cainele latra.
Pisica miauna.
Animalul face un sunet.
```

În exemplul 6.2 clasa `Animal` este clasă de bază, respectiv clasele `Caine` și `Pisica` sunt clase derivate, fiecare având o metodă `vorbeste`(). În funcția principală, creăm o listă de obiecte, fiecare fiind de tip diferit. La runtime, Python determină ce metodă să apeleze în funcție de tipul real al obiectului, demonstrând polimorfismul într-o manieră similară C++.

6.1 Polimorfism în limbaje statice (C++) versus limbaje dinamice (Python)

Observând comparativ cele doua exemple, 6.1 respectiv 6.2, putem deduce o serie de diferențe între C++ și Python în ceea ce privește polimorfisumul.

Polimorfism în C++

C++ este un limbaj de programare cu tipare statice, ceea ce înseamnă că tipul obiectelor și funcțiilor trebuie să fie cunoscut în timpul compilării. Acest lucru determină necesitatea mecanismelor speciale pentru a implementa polimorfismul, cum ar fi funcțiile virtuale și moștenirea. În C++, polimorfismul dinamic este implementat prin intermediul unui mecanism de tabel de funcții virtuale (vtable), unde fiecare clasă derivată care redefinește funcții virtuale are propriul său tabel de funcții care este utilizat la runtime pentru a determina ce funcție trebuie apelată.

Acest mecanism oferă siguranță și performanță în cod, dar necesită o organizare și planificare mai atentă a ierarhiilor de clase și a funcțiilor virtuale.

Polimorfism în Python

Python este un limbaj cu tipare dinamice, ceea ce înseamnă că tipul obiectelor este determinat în timpul execuției, nu în timpul compilării. Prin urmare, polimorfismul este mai natural și simplu în Python. Nu este nevoie de funcții virtuale sau tabele speciale pentru a gestiona polimorfismul dinamic; Python folosește un model numit duck typing. Acesta se bazează pe principiul că, dacă un obiect "se comportă ca o rață" (adică implementează metoda necesară), atunci poate fi tratat ca atare, indiferent de tipul său.

Acest lucru face polimorfismul în Python mult mai flexibil și mai simplu de utilizat. Programatorii nu trebuie să declare moșteniri sau funcții virtuale pentru a permite un comportament polimorfic, ci pur și simplu definesc metode care pot fi apelate pe orice obiect care le implementează.

6.2 Tipuri de polimorfism

Polimorfismul poate fi împărțit în două tipuri principale: polimorfism la compilare (sau polimorfism static) și polimorfism la runtime (sau polimorfism dinamic).

6.2.1 Polimorfism la compilare (polimorfism static)

Polimorfismul la compilare, cunoscut și sub numele de polimorfism static, se manifestă, în limbajul C++ in special, prin faptul că decizia legată de ce metodă să fie apelată este luată în timpul compilării. Exemplele comune includ supraincărcarea funcțiilor și supraincărcarea operatorilor. Supraincărcarea funcțiilor permite mai multor funcții cu același nume să existe în aceeași clasă, dar care diferă prin semnătură (tipuri și/sau număr de parametri). În acest caz, compilatorul determină care variantă a funcției să fie apelată pe baza tipurilor de argumente utilizate în apelul funcției.

De exemplu, într-un limbaj ca C++, putem avea mai multe funcții numite adaugă:

```
1  class Matematica {
2  public:
3      int adauga(int a, int b) {
4          return a + b;
5      }
6
7      double adauga(double a, double b) {
8          return a + b;
9      }
10 };
```

Compilatorul determină care versiune a metodei `adauga` să fie utilizată în funcție de tipul de argumente transmise.

Acest tip de polimorfism este limitat de faptul că deciziile sunt luate în timpul compilării, deci nu poate profita de contextul runtime pentru a ajusta comportamentul. În Python nu există polimorfism static în mod explicit așa cum există în limbaje precum C++.

Exista 2 categorii de polimorfism static, și anume:

- **supraîncarcarea funcțiilor (eng. function overloading)**: Aceasta presupune existența mai multor funcții cu același nume, dar cu parametri diferiți. Compilatorul decide ce funcție să apeleze în funcție de semnătura funcției.
- **supraîncarcarea operatorilor (eng. operator overloading)**: Aceasta permite redefinirea comportamentului operatorilor pentru diferite tipuri de date.

Asptectele legate de supradefinirea funcțiilor și operatorilor vor fi discutate in sub-capitolelele următoare.

6.2.2 Polimorfism la runtime (polimorfism dinamic)

Polimorfismul la runtime, cunoscut și ca `polimorfism dinamic`, se referă la faptul că decizia cu privire la metoda specifică care trebuie apelată este luată la runtime, nu în timpul

compilării. Acesta este realizat în principal prin `funcții virtuale` (în C++) sau `metode suprascrise` (în Python și alte limbaje dinamice). Acest tip de polimorfism permite unei funcții să fie apelată pe un obiect de tipul unei clase de bază, dar implementarea apelată să fie cea specifică clasei derivate, dacă obiectul este de acel tip. Acest lucru este util pentru a crea aplicații mai flexibile, care să răspundă diferitelor tipuri de obiecte într-o manieră dinamică.

Un exemplu simplu este o ierarhie de clase care descriu animale:

```
class Animal {
public:
    virtual void vorbeste() {
        std::cout << "Animalul face un sunet." << std::endl;
    }
};

class Caine : public Animal {
public:
    void vorbeste() override {
        std::cout << "Cainele latra." << std::endl;
    }
};
```

În acest exemplu, funcția `vorbeste()` este marcată ca virtuală în clasa de bază. La runtime, atunci când un obiect de tip `Caine` este apelat printr-un pointer sau o referință de tip `Animal`, funcția corespunzătoare din `Caine` va fi apelată, demonstrând astfel polimorfismul dinamic.

În Python, conceptul de polimorfism dinamic este similar cu cel din C++, dar sintaxa este diferită. În Python, utilizăm metodele din clasele de bază care pot fi supradefinite în clasele derivate, fără a necesita cuvântul cheie `virtual`. De asemenea, nu trebuie să folosim override, dar putem specifica explicit metodele pe care le supradefinim.

Iată cum ar arăta echivalentul în Python pentru codul C++ anterior:

```
class Animal:
    def vorbeste(self):
        print("Animalul face un sunet.")

class Caine(Animal):
    def vorbeste(self):
        print("Cainele latra.")

# Functia principala pentru testare
def main():
    animal = Caine()  # Cream un obiect de tip Caine
    animal.vorbeste()  # Apelam metoda vorbeste() care este
```

```
        ↪ supradefinita in clasa Caine
13
14 if __name__ == "__main__":
15     main()
```

6.3 Supradefinirea (eng. overriding) și supraîncarcarea (eng. overloading) în OOP

Supradefinirea (overriding) și Supraîncarcarea (overloading) sunt doi termeni distincți în programarea orientată pe obiecte.

Supradefinirea (overriding)

Așa cum a fost initial descris la începutul acestui capitol, supradefinirea permite unei clase derivate să redefinească o metodă din clasa de bază cu același nume și semnătură, oferindu-i un comportament specific. Reluăm exemplul de la începutul acestui capitol pentru a exemplifica supradefinirea în C++:

```
1  #include <iostream>
2  using namespace std;
3
4  // Clasa de baza
5  class Animal {
6  public:
7      // Constructor
8      Animal() {}
9
10     // Functie virtuala care poate fi supradefinita in clasele
       ↪ derivate
11     virtual void vorbeste() const {
12         cout << "Animalul face un sunet." << endl;
13     }
14 };
15
16 // Clasa derivata
17 class Caine : public Animal {
18 public:
19     // Supradefinirea functiei vorbeste
20     void vorbeste() const override {
21         cout << "Cainele latra." << endl;
22     }
23 };
24
```

```
25 // Clasa derivata
26 class Pisica : public Animal {
27 public:
28     // Supradefinirea functiei vorbeste
29     void vorbeste() const override {
30         cout << "Pisica miauna." << endl;
31     }
32 };
33
34 int main() {
35     // Crearea obiectelor
36     Animal* animal1 = new Caine();
37     Animal* animal2 = new Pisica();
38
39     // Apelarea functiei vorbeste
40     animal1->vorbeste(); // Output: Cainele latra.
41     animal2->vorbeste(); // Output: Pisica miauna.
42
43     // Eliberarea memoriei
44     delete animal1;
45     delete animal2;
46
47     return 0;
48 }
```

Acest exemplu ilustrează cum supradefinirea permite implementarea diferitelor comportamente pentru funcții în clase derivate, permițând utilizarea aceleași interfețe (`vorbeste()`) pentru tipuri diferite de obiecte (`Caine`, `Pisica`).

Supraîncarcarea (overloading)

Supraîncarcarea permite definirea mai multor funcții sau metode cu același nume, dar cu semnături diferite (adică un număr diferit de parametri sau tipuri diferite de parametri). În C++, poți avea mai multe funcții cu același nume, dar cu parametri diferiți. Compilatorul decide ce funcție să apeleze pe baza semnăturii funcției, după cum urmeaza în următorul exemplu de C++:

Exemplu 6.3 Exemplu, în limbajul C++, de supraîncarcare a functie de adaugare.

```
1 #include <iostream>
2 using namespace std;
3
4 class Calculator {
5 public:
```

```
6       // Supraancarcarea functiei "adauga" pentru diferite tipuri de
            ↪ parametri
7       int adauga(int a, int b) { return a + b; }
8       double adauga(double a, double b) { return a + b; }
9   };
10
11  int main() {
12      Calculator calc;
13      cout << calc.adauga(3, 4) << endl;          // Output: 7
14      cout << calc.adauga(3.5, 4.5) << endl;      // Output: 8.0
15      return 0;
16  }
```

La rularea exemplului 6.3 efectul afisat în consola este următorul:

```
7
8
```

În exemplul de mai sus, funcția `adauga` este supraîncarcată pentru a putea lucra atât cu numere întregi, cât și cu numere zecimale. În funcție de tipul parametrilor, compilatorul decide ce versiune a funcției să apeleze. În Python, supraîncarcarea funcțiilor așa cum este implementată în limbaje precum C++ nu este suportată nativ. Python nu permite definirea mai multor funcții cu același nume dar cu semnături diferite. Totuși, există modalități de a simula acest comportament prin utilizarea argumentelor implicite sau a funcțiilor variabile cum ar fi:

Exemplu 6.4 Exemplu, în limbajul Python, de supraîncarcare a functie de adaugare.

```
1  class Calculator:
2      def adauga(self, a, b=None):
3          if b is None:
4              return a + a  # Daca b nu este specificat, aduna a cu
                   ↪ sine
5          return a + b  # Altfel, aduna a si b
6
7  calc = Calculator()
8
9  print(calc.adauga(3))
10 print(calc.adauga(3, 4))
```

La rularea exemplului 6.4 efectul afisat în consola este următorul:

```
6
7
```

În acest exemplu, metoda `adauga` poate să adune un singur număr la sine sau două numere. Dacă al doilea argument (b) nu este specificat, metoda adună numărul la sine, altfel adună cele două numere.

6.4 Funcții virtuale

Funcțiile virtuale se regăsesc doar în limbajul C++. O funcție virtuală este un concept specific programării orientate pe obiecte (OOP) care permite unei metode definite într-o clasă de bază să fie suprascrisă de o clasă derivată, permițând astfel polimorfismul la runtime. Cu ajutorul funcțiilor virtuale, comportamentul unei metode este determinat la runtime, în funcție de tipul obiectului care invocă funcția, chiar dacă acel obiect este referențiat printr-un pointer sau o referință la clasa de bază.

În cazul unei clase de bază `B` și a unor clase derivate precum `D1`, `D2`, `D3`, etc., care rescriu o metodă `M`, apare întrebarea de cum stabilește compilatorul metoda care va fi apelată. Dacă se utilizează operatorul de rezoluție a scopului (de exemplu, `B::M()`) sau un apel direct al metodei prin intermediul unui obiect (de exemplu, `D1 d; d.M();`), decizia se face în timpul compilării, fiind o `legătură statică` (eng. `early binding`).

Totuși, pot exista situații în care un pointer la clasa de bază poate fi atribuit unui obiect de tipul unei clase derivate sau chiar al clasei de bază, iar în acest caz, compilatorul trebuie să decidă ce metodă va fi invocată în timpul execuției programului. Aceasta este o `legătură dinamică` (eng. `late binding`). Funcțiile care sunt asociate cu legătura dinamică se numesc `funcții virtuale`, iar acestea se marchează cu cuvântul cheie `virtual`. Când o funcție este declarată virtuală într-o clasă de bază și este rescrisă în întregul arbore ierarhic al claselor derivate, aceasta urmează principiul legăturii dinamice.

Funcțiile virtuale prezintă următoarele caracteristici:

- sunt funcții membre non-statice ale unei clase;
- rescrierea funcției virtuale trebuie să respecte prototipul original;
- orice funcție virtuală definită într-o clasă de bază rămâne virtuală și în clasele derivate;
- nu este obligatoriu să rescriem funcțiile virtuale în clasele derivate;
- în cazul în care o funcție virtuală este rescrisă cu un prototip diferit, aceasta va deveni o funcție supradefinită, iar legătura dinamică nu va mai fi aplicată pentru aceste funcții;
- constructorii nu pot fi declarați virtuali, dar destructorii pot fi;
- funcțiile `inline` nu pot fi virtuale.

O clasă care conține o funcție virtuală se mai numește și `clasă polimorfică`. O funcție virtuală își păstrează acest atribut indiferent de câte ori este moștenită. Așadar, dacă o clasă derivată, care a moștenit de la clasa de bază o funcție virtuală, devine o clasă de bază pentru o altă clasă derivată, funcția virtuală poate fi în continuare redefinită.

Dacă o clasă derivată dintr-o clasă de bază nu redefinește funcția virtuală prezentă în clasa de bază, atunci, când un obiect din clasa derivată solicită acces la funcția respectivă, se va folosi funcția definită în clasa de bază.

Într-o ierarhie de clase, accesul la o funcție virtuală, pentru un obiect al unei clase derivate în care nu s-a făcut redefinirea funcției virtuale respective, constă în accesarea

primei redefiniri a sa în ordinea inversă derivării.

Corelarea între funcții virtuale și alte funcții virtuale sau non-virtuale:

- funcțiile virtuale nu pot fi supraîncărcate, dar pot apela astfel de funcții;
- funcțiile virtuale pot apela funcții care nu sunt virtuale și invers;
- funcțiile virtuale pot apela alte funcții virtuale.

În Exemplul 6.5. se declară ierarhia simplă de clase `pozitie → punct → caracter`.

Exemplu 6.5 Exemplu, în limbaj C++, de utilizare a funcțiilor virtuale plecând de la ierarhia de clase `pozitie → punct → caracter`.

```
1  #include <iostream>
2
3  class pozitie     // declarare clasa pozitie
4  {
5  protected: // se permite accesul claselor derivate catre datele
       ↪ membre
6      int x, y;
7  public:
8      pozitie(int =0, int=0);
9      virtual void afisare(); // functia afisare() se declara
           ↪ virtuala
10     void deplasare(int, int);
11 };
12
13 class punct : public pozitie // declararea clasei derivate punct
14 {
15 protected: // se permite accesul claselor derivate catre datele
       ↪ membre
16     int vizibil;
17     int culoare;
18 public:
19     punct(int=0, int=0, int=1);
20     void afisare(); // se redefineste functia virtuala afisare()
21     void deplasare(int, int);
22 };
23
24 class caracter:public punct // declararea clasei derivate caracter
25 {
26         char c;
27 public:
28         caracter(int, int, int, char);
29         void afisare(); // se redefineste functia virtuala afisare
```

```
            ↪ ()
30          void deplasare(int, int);
31  };
32
33  pozitie::pozitie(int abs, int ord)
34  {
35      std::cout<<"\nConstructor pozitie:"<<this;
36      x=abs;      y=ord;
37      std::cout<<" x="<<x<<" y="<<y;
38  }
39
40  void pozitie::afisare()
41  {
42      std::cout<<"\nPozitie:"<<this;
43      std::cout<<" x="<<x<<" y="<<y<<std::endl;
44  }
45
46  void pozitie::deplasare(int dx, int dy)
47  {
48      x+=dx;      y+=dy;
49      std::cout<<"\nDeplasare pozitie:";
50      afisare();
51  }
52
53  punct::punct(int abs, int ord, int cul):pozitie(abs, ord)
54  {
55      std::cout<<"\nConstructor punct:"<<this;
56      culoare=cul;      vizibil=0;
57      std::cout<<" x="<<x<<" y="<<y<<" culoare="<<culoare<<" vizibil
            ↪ ="<<vizibil;
58  }
59
60  void punct::afisare()
61  {   std::cout<<"\nPunct:"<<this;
62      std::cout<<" x="<<x<<" y="<<y<<" culoare="<<culoare<<" vizibil
            ↪ ="<<vizibil<<std::endl;
63  }
64
65  void punct::deplasare(int dx, int dy)
66  {
67      x+=dx;      y+=dy;
```

```
68      std::cout<<"\nDeplasare punct:";
69      afisare();
70  }
71  caracter::caracter(int abs, int ord, int cul, char ch):punct(abs,
        ↪ ord, cul)
72  {
73      std::cout<<"\nConstructor caracter:"<<this;
74      c=ch;
75      std::cout<<" x="<<x<<" y="<<y<<" culoare="<<culoare<<" vizibil
            ↪ ="<<vizibil;
76      std::cout<<" caracter=:"<<c;
77  }
78
79  void caracter::afisare()
80  {
81      std::cout<<"\nCaracter:"<<this;
82      std::cout<<" x="<<x<<" y="<<y<<" culoare="<<culoare<<" vizibil
            ↪ ="<<vizibil;
83      std::cout<<" caracter="<<c<<std::endl;
84  }
85
86  void caracter::deplasare(int dx, int dy)
87  {
88      x+=dx;      y+=dy;
89      std::cout<<"\nDeplasare caracter:";
90      afisare();
91  }
92
93  int main()
94  {
95      // se declara obiecte de tip clasa
96      pozitie p1(1, 1);
97      punct p2(2, 2, 1);
98      caracter c1(3, 3, 3, '*');
99
100     // se declara pointeri la clase cu initializare, intre tipul
            ↪ pointerilor si tipul obiectului
101     // existand corespondenta
102     pozitie* p_pozitie=&p1;
103     punct * p_punct=&p2;
104     caracter * p_car=&c1;
```

```
105
106     // apeluri ale functiilor membre
107     p_pozitie->afisare();
108     p_pozitie->deplasare(2, 2);
109     p_punct->afisare();
110     p_punct->deplasare(3, 3);
111     p_car->afisare();
112     p_car->deplasare(5, 5);
113
114     p_pozitie=p_punct; // atribuire admisa, prin conversia
        ↪ implicita punct*->pozitie*;
115     p_pozitie->afisare();
116     p_pozitie->deplasare(3, 3);
117
118     p_pozitie=p_car; // atribuire admisa, prin conversia implicita
        ↪   caracter*->pozitie*;
119     p_pozitie->afisare();
120     p_pozitie->deplasare(3, 3);
121     return 0;
122 }
```

La rularea exemplului 6.5 efectul afisat în consola este următorul:

```
Constructor pozitie:0x7fff1e6b2320 x=1 y=1
Constructor pozitie:0x7fff1e6b2330 x=2 y=2
Constructor punct:0x7fff1e6b2330 x=2 y=2 culoare=1 vizibil=0
Constructor pozitie:0x7fff1e6b2350 x=3 y=3
Constructor punct:0x7fff1e6b2350 x=3 y=3 culoare=3 vizibil=0
Constructor caracter:0x7fff1e6b2350 x=3 y=3 culoare=3 vizibil=0 caracter=:*
Pozitie:0x7fff1e6b2320 x=1 y=1

Deplasare pozitie:
Pozitie:0x7fff1e6b2320 x=3 y=3

Punct:0x7fff1e6b2330 x=2 y=2 culoare=1 vizibil=0

Deplasare punct:
Punct:0x7fff1e6b2330 x=5 y=5 culoare=1 vizibil=0

Caracter:0x7fff1e6b2350 x=3 y=3 culoare=3 vizibil=0 caracter=*

Deplasare caracter:
```

```
Caracter:0x7fff1e6b2350 x=8 y=8 culoare=3 vizibil=0 caracter=*

Punct:0x7fff1e6b2330 x=5 y=5 culoare=1 vizibil=0

Deplasare pozitie:
Punct:0x7fff1e6b2330 x=8 y=8 culoare=1 vizibil=0

Caracter:0x7fff1e6b2350 x=8 y=8 culoare=3 vizibil=0 caracter=*

Deplasare pozitie:
Caracter:0x7fff1e6b2350 x=11 y=11 culoare=3 vizibil=0 caracter=*
```

Pointerul `pozitie* p_pozitie` permite trecerea de la un tip mai general (pozitie) la un tip mai specializat (punct sau caracter), fiind o operație sigură, numită *upcast*. Se pierde însă informația despre tipul obiectului, iar compilatorul poate gestiona obiectul doar prin intermediul pointerului sau referinței la clasa generală; astfel, se va apela metoda `deplasare()` din clasa `pozitie`!

Motivul acestui comportament este rezolvarea apelului de funcție la compilare și link-editare, tehnică numită *early binding*. Funcția `deplasare()` va fi apelată corespunzător tipului pointerului.

Din exemplul prezentat se observă că apelul funcției `afisare()` se face corespunzător obiectului declarat, chiar dacă pointerul este de tipul clasei de bază, datorită faptului că funcția a fost declarată virtuală. Mecanismul funcțiilor virtuale implementează tehnica *late binding*: rezolvarea apelului de funcție în timpul execuției programului. În acest scop, compilatorul inserează în program cod prin care se determină, la execuție, care este definiția corectă a funcției.

Dacă se înlocuiește declarația funcției `deplasare()` a clasei `pozitie` cu declarația:

```
1 virtual void deplasare(int, int);
```

se va observa în urma execuției programului că și apelul acesteia se face corespunzător tipului obiectului și nu tipului pointerului. În această situație, corespunzător secvenței:

```
1     p_pozitie=p_punct;
2     p_pozitie->afisare();
3     p_pozitie->deplasare(3, 3);
4     p_pozitie=p_car;
5     p_pozitie->afisare();
6     p_pozitie->deplasare(3, 3);
```

din funcţia main(), se va afişa:

```
Punct:   0xffe6  x=5 y=5 culoare=1 vizibil=0
```

```
Deplasare punct:
Punct:   0xffe6  x=8 y=8 culoare=1 vizibil=0

Caracter:0xffd8 x=8 y=8 culoare=3 vizibil=0  caracter=*

Deplasare caracter:
Caracter:0xffd8 x=11 y=11 culoare=3 vizibil=0  caracter=*
```

În clasele derivate, dacă se folosește un prototip diferit pentru declararea unei funcții care este virtuală în clasa de bază, atunci funcția va fi considerată supradefinită și legătura dinamică nu va mai fi aplicată. Deși compilatorul acceptă această schimbare, va afișa un mesaj de atenționare pentru a semnala faptul că noua declarație ascunde funcția virtuală existentă.

De exemplu, în clasa `caracter`, declarația funcției `deplasare` poate fi înlocuită cu următoarea declarație:

```
1 void deplasare();
```

cu definiţia:

```
1 void caracter::deplasare()
2 {
3       x+=1;       y+=1;
4       std::cout<<"\nDeplasare caracter:";
5       afisare();
6 }
```

În acest caz, apelul funcţiei se va face:

```
1 p_car->deplasare(); // functia este fara parametri
```

Pentru secvenţa:

```
1       p_pozitie=p_car;
2       p_pozitie->afisare();
3       p_pozitie->deplasare(3, 3);
```

se va afişa:

```
Caracter:0xffd8 x=8 y=8 culoare=3 vizibil=0  caracter=*

Deplasare punct:
Caracter:0xffd8 x=11 y=11 culoare=3 vizibil=0  caracter=*
```

Se poate observa că funcţia selectată deplasare() apelată, datorită existenţei parametrilor din apel, este funcţia moştenită de la clasa punct. Apelul :

```
1 p_pozitie->deplasare();
```

va fi sancţionat cu mesaj de eroare, se cer parametri pentru funcţia deplasare(), deoarece pointerul la pozitie încearcă să acceseze funcţia virtuală. Nu este obligatorie redefinirea funcţiilor virtuale în clasele derivate, astfel că, dacă eliminăm declaraţia funcţiei deplasare() din clasa caracter, se va apela redefinirea din ultima clasă ierarhic moştenită, deci apelul:

```
1 p_pozitie->deplasare(3, 3);
```

va avea ca efect afişarea:

```
Deplasare punct:
Caracter:0xffd8 x=11 y=11 culoare=3 vizibil=0  caracter=*
```

Având în vedere că folosirea funcţiilor virtuale presupune un consum de memorie suplimentar şi operaţii suplimentare care au ca urmare creşterea timpului de execuţie, se recomandă evitarea utilizării nejustificate a funcţiilor virtuale.

Se ştie că fiecare funcţie membru are ataşat pointerul this. În cazul funcţiilor virtuale se utilizează o tabelă de pointeri către funcţii, care se mai numeşte şi tabelă de funcţii virtuale numită `VTABLE`. Fiecare clasă, care posedă o funcţie virtuală, are ataşată o astfel de tabelă. Tabela are o intrare pentru fiecare funcţie virtuală, incluzând şi pe cele moştenite de la clasa de bază. O astfel de intrare constă dintr-un pointer care pointează către funcţia apelată de funcţia virtuală corespunzătoare. Tabela conţine adresele definiţiilor funcţiilor membre virtuale. Toate instanţele (obiectele) unei astfel de clase conţin o dată membru suplimentară, numită `VPTR`, care nu este altceva decât un pointer către tabela `VTABLE`. La apelul unei funcţii virtuale prin intermediul unui pointer/referinţă la clasa de bază compilatorul inserează cod prin care, la execuţie, dereferenţiind `VPTR`, se accesează `VTABLE` şi se găseşte adresa funcţiei.

Exemplu 6.6 Exemplu, în limbaj C++, de utilizare a funcțiilor virtuale plecând de la ierarhia de clase `pozitie → punct → caracter`.

```
1  #include <iostream>
2
3  class NoVirtual
4  {
5      int a;
6      public:
7      void x() const{};
8      int i() const{
9      return 1;
10     };
11 };
12
13 class OneVirtual
```

```
14 {
15     int a;
16     public:
17     virtual void x(){};
18     int i() const{
19     return 1;
20     };
21 };
22
23 class TwoVirtual
24 {
25     int a;
26     public:
27     virtual void x(){ };
28     virtual int i()
29 {     return 1;     }
30 };
31
32 class ThreeVirtual
33 {
34     int a;
35     public:
36     virtual void x(){};
37     virtual int i()
38 {     return 1;    };
39     virtual double r()
40 {    return 1;    };
41 };
42
43
44 int main()
45 {
46     std::cout << "int:" << sizeof(int) << std::endl;
47     std::cout << "void*:" << sizeof(void*) << std::endl;
48     std::cout << "NoVirtual:" << sizeof(NoVirtual) << std::endl;
49     std::cout << "OneVirtual:" << sizeof(OneVirtual) << std::endl;
50     std::cout << "TwoVirtual:" << sizeof(TwoVirtual) << std::endl;
51     std::cout << "ThreeVirtual:" << sizeof(ThreeVirtual) << std::
       ↪ endl;
52     return 0;
53 }
```

La rularea exemplului 6.6 efectul afisat în consola este următorul:

```
int:4
void*:8
NoVirtual:4
OneVirtual:16
TwoVirtual:16
```

Se observă că:

- fără funcţii virtuale, dimensiunea obiectului este exact dimensiunea lui `int` (dată membru);
- prin apariţia unei funcţii virtuale, dimensiunea obiectului a crescut cu dimensiunea lui `void*` (VPTR);
- nu există diferenţă de dimensiune între clase cu una sau mai multe funcţii virtuale; pentru o clasă se generează o singură tabelă VTABLE, care conţine adresele tuturor funcţiilor membre virtuale; fiecare instanţă conţine un singur pointer VPTR.

Pentru fiecare instanţă (obiect) a unei clase, VPTR trebuie iniţializat înainte de a se apela vreo funcţie virtuală; din acest motiv constructorii **NU** pot fi funcţii virtuale. Deoarece, într-o ierarhie de clase, destructorii se apelează în ordine inversă ordinii de apel a constructorilor (iar VPTR este deja iniţializat), este recomandat ca destructorii să fie funcţii virtuale (un destructor ne-virtual semnalează intenţia de a nu folosi această clasă drept clasă de bază).

Exemplu 6.7 Exemplu, în limbaj C++, de utilizare a funcțiilor virtuale în constructori și destructori.

```
1  #include <iostream>
2
3  class Baza1
4  {
5      public:
6          ~Baza1()
7          {
8              std::cout << "~Baza1()" << std::endl;
9          };
10 };
11
12 class Baza2
13 {
14     public:
15         virtual ~Baza2()
16         {
17             std::cout << "~Baza2()" << std::endl;
18         };
```

```
19 };
20
21 class Derivata1 : public Baza1
22 {
23     public:
24         ~Derivata1()
25         {
26             std::cout << "~Derivata1()" << std::endl;
27         };
28 };
29
30 class Derivata2 : public Baza2
31 {
32     public:
33         ~Derivata2()
34         {
35             std::cout << "~Derivata2()" << std::endl;
36         };
37 };
38
39 int main()
40 {
41     Baza1* pb1 = new Derivata1;
42     delete pb1;
43
44     Baza2* pb2 = new Derivata2;
45     delete pb2;
46     return 0;
47 }
```

La execuţia programului din exemplul 6.7. se va afişa:

```
~Baza1()
~Derivata2()
~Baza2()
```

Se observă că `delete pb1` apelează doar destructorul clasei de bază, în timp ce `delete pb2` apelează, corect, destructorul clasei derivate, urmat de destructorul clasei de bază. Practic, deşi destructorul nu se moşteneşte, destructorul virtual este supraîncărcat în clasa derivată! Dacă supraîncărcarea unei funcţii virtuale necesită extinderea versiunii din clasa de bază (adică adăugarea de funcţionalitate definiţiei deja existente), se poate proceda ca în exemplul 6.8.

Exemplu 6.8 Exemplu, în limbaj C++, de utilizare a funcțiilor virtuale în constructori și destructori.

```
1  #include <iostream>
2
3  class forma
4  {
5      public:
6          virtual void redimensioneaza(int dx, int dy)
7          {
8              std::cout<<"\nredimensioneaza- forma\n";
9          }
10 };
11
12 class patrat: public forma
13 {
14     public:
15         void redimensioneaza (int dx, int dy)
16         {
17             forma::redimensioneaza(dx, dy); //legatura statica!
18              //adauga functionalitate...
19      std::cout<<"\nredimensioneaza - patrat\n";
20         }
21 };
22
23 int main()
24 {
25     forma* ps=new patrat;
26     ps->redimensioneaza(1,1);
27     return 0;
28 }
```

La execuţie programului din exemplul 6.8 se va afişa:

```
redimensioneaza- forma
redimensioneaza - patrat
```

Se poate observa că, pointerul la clasa `forma ps` este folosit pentru alocarea unui obiect de tip patrat, apelul funcţiei `redimensioneaza()` fiind făcut conform tipului obiectului alocat, nu al pointerului. Schimbarea specificaţiilor de acces la o funcţie virtuală nu este recomandat. Se consideră exemplul 6.9

Exemplu 6.9 Exemplu, în limbaj C++, de schimbare inadecvata a specificațiilor de acces la o funcție virtuală.

```
1  #include <iostream>
```

```
2
3  class baza
4  {
5      public:
6          virtual void vorbeste()
7          {
8              std::cout << "baza" << std::endl;
9          }
10 };
11
12 class derivata: public baza
13 {
14     void vorbeste()
15     {
16         //in absenta unui specificator de acces, vorbeste() este
               ↪ declarat private
17         std::cout << "derivata" << std::endl;
18     };
19 };
20
21 int main()
22 {
23     //...
24     derivata d;
25     baza* pb = &d;
26     pb->vorbeste(); //OK, se apeleaza derivata::vorbeste()
27     return 0;
28 }
```

La execuţie programului din exemplul 6.9 se va afişa:

```
derivata
```

Deoarece `vorbeste()` este funcţie virtuală, legarea târzie împiedică compilatorul să detecteze apelul unei funcţii ne-publice.

6.5 Funcţii virtuale pure. Clase abstracte.

Există multe cazuri în care o clasă de bază nu poate defini suficient un obiect pentru a permite să fie creată o funcţie virtuală în acea clasă. Altfel spus, este posibil să nu existe o definiţie semnificativă a funcţiei virtuale în clasa de bază. Un astfel de deziderat se poate înlătura în C++ folosind funcţii virtuale pure.

O funcţie virtuală pură este o funcţie virtuală pentru care nu se declară o implementare (se moşteneşte doar interfaţa). Sintaxa declarării unei funcţii virtuale pure este:

```
1 virtual tip_returnat nume(...) = 0;
```

Din punct de vedere practic, compilatorul rezervă în VTABLE câte o locaţie "goală" pentru fiecare funcţie virtuală pură. O clasă care conţine cel puţin o funcţie virtuală pură este o clasă abstractă şi nu poate fi instanţiată.

Clasele care moştenesc clase abstracte trebuie să implementeze toate funcţiile virtuale pure; în caz contrar, sunt şi ele clase abstracte. Aceasta deoarece se copiază VTABLE din clasa de bază şi se modifică doar adresele funcţiilor virtuale supraîncărcate. De aici rezultă şi faptul că, dacă o funcţie virtuală nu este supraîncărcată, clasa derivată are acces la definiţia funcţiei din clasa imediat superioară în ierarhie! Cu alte cuvinte, se moşteneşte întotdeauna interfaţa (prototipul) şi, opţional, implementarea. Ultima supraîncărcare a unei funcţii virtuale se numeşte *eng. last overridden*.

Un destructor poate fi virtual pur. Însă destructorul unei clase derivate trebuie să poată apela destructorul clasei de bază. Din acest motiv, un destructor virtual pur trebuie să aibă o implementare (vidă).

```
1 class Interface
2 {
3 public:
4     virtual void Open() = 0;
5     virtual ~Interface() = 0;
6 };
7
8 Interface::~Interface(){} // implementare vida
```

Deoarece o clasă abstractă conţine funcţii, pentru care nu există definiţii (funcţii virtuale pure), nu se pot crea obiecte ale sale. Totuşi se pot crea pointeri şi referinţe către aceste clase. De aceea clasele abstracte admit polimorfismul în timpul execuţiei, care constă în alegerea funcţiei virtuale corespunzătoare de către pointerii clasei de bază. În C++, o clasă de bază poate fi folosită pentru a defini natura unei interfeţe cu o clasă generală. Fiecare clasă derivată va introduce operaţiile specifice solicitate de tipurile de date folosite de tipul derivat. Definind o clasă abstractă, vom reuşi să stabilim un mod unic de comunicare cu utilizatorul. Unui astfel de mod de comunicare îi va fi ataşat un set de metode (funcţii membre) caracteristice claselor derivate. De aceea se mai spune că utilizarea funcţiilor virtuale, a claselor abstracte şi a polimorfismului în timpul execuţiei constituie cele mai puternice şi flexibile căi de a obţine o interfaţă unică cu metode multiple. Cu aceasta se realizează o ierarhizare de la general la specific.

În 6.10 este prezentată declararea unei clase abstracte şi folosirea ei în procesul derivării.

Exemplu 6.10 Exemplu, în limbaj C++, de utilizare a clase abstracte.

```
1 #include <iostream>
2
```

```
3  class baza
4  {
5      public:
6          virtual void func()=0;
7          virtual ~baza()=0;
8
9  };
10
11 baza::~baza()
12 {} // implementare vida
13
14 class derivata1: public baza
15 {
16     void func()
17     {
18         std::cout<<"\nderivata1";
19     }
20 };
21
22 class derivata2: public baza
23 {
24     void func()
25     {
26         std::cout<<"\nderivata2";
27     }
28 };
29
30 int main()
31 {
32     baza b; // Eroare !!- nu se pot declara obiecte de tipul unei
       ↪ clase abstracte
33     derivata1 d1;
34     derivata2 d2;
35
36     baza* pb ; // se declara pointer la tipul baza
37     pb= &d1;
38     pb->func(); // referirea unui obiect de tip derivata1 prin
       ↪ pointer la tipul base
39
40     pb = &d2;
41     pb->func(); // referirea unui obiect de tip derivata2 prin
```

```
        ↪ pointer la tipul baza
42      return 0;
43  }
```

La execuţie programului din exemplul 6.10 se va afişa:

```
main.cpp: In function ‘int main()’:
main.cpp:34:10: error: cannot declare variable ‘b’ to be of abstract type ‘baza’
   34 |     baza b;     // Eroare !!- nu se pot declara obiecte de tipul unei clase
      |          ^
```

7. Conceptul de încapsulare

Fundamentele încapsulării
Specificatori de acces (public, privat, protejat)
Get și set
Decoratorii de proprietate (Python)

7.1 Fundamentele încapsulării

Încapsularea este unul dintre cei patru piloni ai programării orientate pe obiect (OOP). Încapsularea se referă la ascunderea detaliilor de implementare ale unei clase și expunerea doar a funcționalităților esențiale. Acest lucru ajută la reducerea complexității și la creșterea modularității codului.

Principalele beneficii ale încapsulării sunt următoarele:

- **reducerea complexității:** prin ascunderea detaliilor de implementare, încapsularea permite dezvoltatorilor să se concentreze pe interfața publică a clasei, fără a se preocupa de complexitatea implementării interne.
- **modularitate:** încapsularea permite crearea de module independente care pot fi dezvoltate și testate separat, facilitând întreținerea și extinderea codului.
- **protecția datelor:** restricționarea accesului direct la date ajută la prevenirea modificărilor neautorizate și asigură integritatea acestora, esențială pentru construirea unui software robust și sigur.
- **flexibilitate și evoluție:** schimbările în implementarea internă a unei clase pot fi efectuate fără a afecta codul care utilizează acea clasă, atâta timp cât interfața publică rămâne constantă.

Prin urmare, încapsularea nu doar că promovează bunele practici de programare, dar contribuie semnificativ la crearea unor aplicații mai clare, mai sigure și mai ușor de întreținut.

În exemplul 7.1 este prezentat un exemplu simplu de încapsulare în limbajul C++.

Exemplu 7.1 Exemplu, în limbaj C++, de încapsulare a datelor.

```
1 class ClasaMea {
2 private:
```

```
3       int __date; // Membru privat, accesibil doar din interiorul
            ↪ clasei
4
5   public:
6       // Set public pentru date
7       void setDate(int valoare) { __date = valoare; }
8
9       // Get public pentru date
10      int getDate() const { return __date; }
11  };
12
13  int main()
14  {
15      ClasaMea c;
16  }
```

În acest exemplu, `date` este un atribut privat, indicat printr-un prefix dublu (două linii de subliniere). Acest prefix sugerează că atributul nu ar trebui accesat direct din afara clasei. Metodele setDate și getDate sunt publice și oferă acces controlat la datele private ale clasei.

Echivalent, în Python, conceptul de încapsulare se poate prezenta după cum urmează:

Exemplu 7.2 Exemplu, în limbaj Python, de încapsulare a datelor.

```
1   class ClasaMea:
2       def __init__(self, valoare):
3           self.__date = valoare  # Membru privat, indicat prin
                ↪ prefixul dublu
4
5       def get_date(self, valoare):
6           self.__date = valoare  # Set public
7
8       def set_date(self):
9           return self.__date  # Get public
```

În acest exemplu, `__date` este un membru privat care nu poate fi accesat direct din afara clasei. Accesul se face prin metodele `set_date` și `get_date`, asigurând protecția și integritatea datelor. Metodele `set_date` și get_date permit utilizatorilor clasei să interacționeze cu datele private fără a expune detaliile interne ale implementării.

7.2 Specificatori de acces (public, privat, protected)

Specificatorii de acces sunt un aspect crucial al programării orientate pe obiect (OOP) și sunt utilizați pentru a controla vizibilitatea și accesibilitatea membrilor (atribute și metode) unei clase. În mod obișnuit, aceștia sunt folosiți pentru a proteja datele interne ale unei clase

și pentru a oferi o interfață clară și sigură pentru interacțiunea cu obiectele. Specificatorii de acces pot fi clasificați în trei categorii: public, privat și protejat.

Specificatorul de access `public`

Membrii declarați ca publici sunt accesibili din orice loc în program. Nu există restricții privind accesul la acești membri. Acest speficator permite accesul la membri de care este nevoie pentru interacțiunea cu obiectele, oferind funcționalități necesare utilizatorilor clasei.

Exemplu în C++:

```
1 class ClasaMea {
2 public:
3     int publicData; // Membru public, accesibil din afara clasei
4 };
```

Modul de utilizare este următorul:

```
1 ClasaMea obj;
2 obj.publicData = 10; // Acces direct la membrul public
```

În Python, specificatorii de acces nu sunt la fel de riguroși ca în C++ și, din acest motiv, comportamentul poate părea similar între membrii publici și privați. Cu toate acestea, Python utilizează convenții de denumire pentru a indica vizibilitatea membrilor și a aplica restricții. Astfel, specificatorul de acces public este unul implicit, deci nu trebuie mentionat în mod particular termenul `public`.

```
1 class ClasaMea:
2     def __init__(self, data):
3         self.public_data = data  # Membru public, accesibil din
              ↪ afara clasei
```

Modul de utilizare este următorul:

```
1 obj = ClasaMea(10)
2 print(obj.public_data)  # Acces direct la atributul public
```

Specificatorul de access `privat`

Membrii declarați ca privați sunt accesibili doar din interiorul clasei în care sunt definiți. Ei nu pot fi accesați direct din afara clasei, nici de către clasele derivate. În acest fel, speficatorul private protejează datele și funcționalitățile interne ale clasei, asigurând că modificările se fac doar prin metode controlate (set/get). Iată un exemplu de utilizare a specificatorului `private`, în C++

```
1 class ClasaMea {
2 private:
3     int privateData; // Membru privat, accesibil doar din
        ↪ interiorul clasei
```

```
4
5 public:
6     void setPrivateData(int data) { privateData = data; } // Set
      ↪ public
7     int getPrivateData() const { return privateData; } // Get
      ↪ public
8 };
```

Modul de utilizare este următorul:

```
1 ClasaMea obj;
2 obj.setPrivateData(20); // Setare a valorii prin metoda publica
3 int value = obj.getPrivateData(); // Accesarea valorii prin metoda
  ↪ publica
```

În Python, prefixul dublu `__` (două linii de subliniere) înainte de un nume de variabilă sau metodă are un rol specific și este asociat cu o tehnică numită gestionarea numelui (eng. name mangling). Această tehnică este utilizată pentru a face variabilele și metodele mai dificil de accesat accidental din afara clasei, adică pentru a indica că acestea sunt destinate să fie private. Scopul prefixului dublu `__` este de a evita coliziunile de nume între clasele de bază și cele derivate și de a ascunde atributele private de utilizatorii din afara clasei.

```
1 class ClasaMea:
2     def __init__(self, data):
3         self.__private_data = data  # Membru privat
4
5     def get_private_data(self):
6         return self.__private_data
```

În acest exemplu, `__private_data` este un atribut privat, iar prefixul `__` semnalează faptul că acesta este destinat să fie utilizat doar în interiorul clasei `ClasaMea`.

Python transformă numele atributelor care încep cu `__` prin adăugarea unui prefix care include numele clasei în care a fost definit atributul. Aceasta face ca atributul să fie mai greu accesibil din afara clasei, dar nu îl face complet inaccesibil.

```
1 class ClasaMea:
2     def __init__(self, data):
3         self.__private_data = data  # Membru privat
4
5     def get_private_data(self):
6         return self.__private_data
7
8 # Utilizare
9 obj = ClasaMea(42)
```

```
print(obj.get_private_data())  # Acces corect la __private_data prin metoda publica

# Incercarea de a accesa direct atributul __private_data va esua
# print(obj.__private_data)  # Va genera o eroare AttributeError

# Totusi, putem accesa atributul privat folosind name mangling
print(obj._ClasaMea__private_data)  # Acces direct, dar nu recomandat
```

În exemplul anterior, `__private_data` este transformat de Python în `_MyClass__private_data`. Aceasta înseamnă că numele atributului este modificat pentru a include numele clasei ca prefix, ceea ce îl face mai puțin accesibil accidental. Chiar dacă accesul direct la `_MyClass__private_data` este posibil, acesta nu este recomandat, deoarece este considerat o practică nepotrivită și poate duce la probleme de întreținere a codului.

Specificatorul de access `protected`

Membrii declarați ca protejați sunt accesibili din interiorul clasei în care sunt definiți și din clasele derivate. Nu sunt accesibili din afara ierarhiei de moștenire. Principalul avantaj al utilizării specificatorului protected este acela că permite extensia și personalizarea funcționalităților de către clasele derivate, protejând în același timp datele de accesul direct din exterior. Iată un exemplu de utilizare a specificatorului `protected`, în c++

```
class ClasaDeBaza {
protected:
    int protectedData; // Membru protejat, accesibil din clasele derivate

public:
    void setProtectedData(int data) { protectedData = data; }
};

class ClasaDerivata : public ClasaDeBaza {
public:
    void showProtectedData() {
        // Acces la membrul protejat din clasa derivata
        std::cout << protectedData << std::endl;
    }
};
```

Modul de utilizare este următorul:

```
ClasaDerivata obj;
```

```
2 obj.setProtectedData(40); // Setare a valorii prin metoda publica
    ↪ din clasa de baza
3 obj.showProtectedData(); // Accesarea valorii din clasa derivata
```

În Python, membrii protejați sunt indicați printr-un prefix simplu (_). Aceasta este o convenție care sugerează că membrii nu ar trebui să fie accesați din afara clasei sau din clasele derivate, deși nu există restricții stricte pentru accesul acestora. Prefixul simplu _ este utilizat pentru a indica că un atribut sau metodă este destinat să fie considerat `protejat`, adică ar trebui să fie accesat doar din interiorul clasei și al claselor derivate, dar fără a impune restricții stricte. Membrii cu un prefix simplu _ sunt mai puțin ascunși decât cei cu prefix dublu __.

```
1 class ClasaDeBaza:
2     def __init__(self, data):
3         self._protected_data = data  # Membru protejat, indicat
            ↪ prin prefixul simplu
4
5 class ClasaDerivata(ClasaDeBaza):
6     def show_protected_data(self):
7         print(self._protected_data)  # Acces la atributul protejat
            ↪  din clasa derivata
8
9 # Utilizare
10 obj = ClasaDerivata(40)
11 obj.show_protected_data()  # Accesarea valorii din clasa derivata
12
13 # Accesul direct la _protected_data din afara clasei este posibil,
    ↪  dar nu este recomandat
14 print(obj._protected_data)  # Acces direct, dar nu este recomandat
```

În concluzie, membrii publici sunt accesibili din orice loc în program, neexistând restricții; membrii privați sunt denumiți cu prefix dublu (__) și sunt ascunși prin name mangling, accesul direct din afara clasei este descurajat, dar nu imposibil; membrii protected sunt denumiți cu prefix simplu (_). Se recomandă accesul doar din interiorul clasei și din clasele derivate, dar accesul direct din afara clasei este posibil.

7.3 Get și Set

Get și Set sunt metode utilizate pentru a accesa și modifica valorile atributelor unei clase în cadrul programării orientate pe obiect (OOP). Aceste metode sunt considerate o practică bună de programare, deoarece permit controlul asupra modului în care valorile sunt citite sau modificate, protejând datele și oferind o interfață clară pentru manipularea acestora.

Get este o metodă utilizată pentru a obține (citi) valoarea unui atribut dintr-o clasă. Set

este o metodă utilizată pentru a seta (modifica) valoarea unui atribut dintr-o clasă. Get și set oferă un control mai fin asupra accesului la date, permițând validarea datelor, logare, sau alte acțiuni la citirea sau modificarea valorilor atributelor.

În C++, get și set sunt metode publice care sunt utilizate pentru a accesa și modifica atributele private ale unei clase. Acest lucru se face pentru a proteja datele interne și pentru a asigura că acestea sunt utilizate într-un mod controlat.

Exemplu 7.3 Exemplu, în limbaj C++, de utilizare a get și set.

```
1  #include <iostream>
2
3  class ClasaMea {
4  private:
5      int varsta;  // Atribut privat
6
7  public:
8      // Set pentru varsta
9      void seteazaVarsta(int v) {
10         if (v > 0) {
11             varsta = v;
12         } else {
13             std::cout << "Varsta trebuie sa fie pozitiva!" << std
                   ↪ ::endl;
14         }
15     }
16
17     // Get pentru varsta
18     int obtineVarsta() const {
19         return varsta;
20     }
21 };
22
23 int main() {
24     ClasaMea persoana;
25     persoana.seteazaVarsta(25);  // Setare prin set
26     std::cout << "Varsta: " << persoana.obtineVarsta() << std::
          ↪ endl;  // Accesare prin get
27
28     return 0;
29 }
```

La rularea exemplului 7.3 efectul afisat în consola este următorul:

```
Varsta: 25
```

În exemplul de mai sus, `varsta` este un atribut privat. Utilizăm metoda `seteazaVarsta` pentru a seta valoarea variabilei varsta și metoda `obtineVarsta` pentru a obține valoarea acesteia. Astfel, valorile sunt controlate, iar datele sunt protejate.

Principalele avantaje ale Get și Set în C++ sunt:

- `protecția datelor`: prin utilizarea set și get, atributele interne sunt protejate de accesul direct, reducând șansele de corupere a datelor.
- `validarea datelor`: în set, putem valida valorile înainte de a le atribui atributelor, asigurându-ne că acestea sunt corecte.
- `controlul accesului`: permite controlul asupra modului în care sunt citite sau modificate datele.
- `încapsularea`: get și set fac parte din încapsularea datelor, promovând o bună separare a logicii interne de interfața publică a clasei.

Principalele dezavantaje ale Get și Set în C++sunt:

- `cod suplimentar`: crearea de get și set pentru fiecare atribut poate duce la o creștere a codului, făcând clasa mai greu de întreținut.
- `performanță`: în anumite cazuri, utilizarea get și set poate introduce o îngreunare a performanței (deși, de obicei, este neglijabilă).
- `complexitate`: pentru clase simple, utilizarea get și set poate adăuga o complexitate inutilă.

În Python, get și set sunt mai puțin utilizați, deoarece limbajul oferă o caracteristică denumită `proprietăți` (properties), care permite accesul controlat la atribute într-un mod mai natural. Proprietățile permit definirea de metode get și set, dar atributele pot fi accesate ca și cum ar fi variabile membre simple.

În continuare, exemplul 7.4 descrie utilizarea get și set tradiționali în Python, respectiv exemplul 7.5 va prezenta modul de utilizare al decoratorului `@property`.

Exemplu 7.4 Exemplu, în limbaj Python, de utilizare a get și set în mod tradițional.

```
1  class ClasaMea:
2      def __init__(self, varsta):
3          self.__varsta = varsta  # Atribut privat
4
5      # Get pentru varsta
6      def obtine_varsta(self):
7          return self.__varsta
8
9      # Set pentru varsta
10     def seteaza_varsta(self, varsta):
11         if varsta > 0:
12             self.__varsta = varsta
13         else:
14             print("Varsta trebuie sa fie pozitiva!")
```

```
15
16 # Utilizare
17 persoana = ClasaMea(25)
18 print(persoana.obtine_varsta())  # Accesare prin get
19 persoana.seteaza_varsta(30)  # Setare prin set
20 print(persoana.obtine_varsta())
```

La rularea exemplului 7.4 efectul afisat în consola este următorul:

```
25
30
```

Exemplu 7.5 Exemplu, în limbaj Python, de utilizare a get și set, folosind decoratorii de proprietăți.

```
1  class ClasaMea:
2      def __init__(self, varsta):
3          self.__varsta = varsta  # Atribut privat
4
5      @property
6      def varsta(self):
7          return self.__varsta
8
9      @varsta.setter
10     def varsta(self, varsta):
11         if varsta > 0:
12             self.__varsta = varsta
13         else:
14             print("Varsta trebuie sa fie pozitiva!")
15
16 # Utilizare
17 persoana = ClasaMea(25)
18 print(persoana.varsta)  # Acces direct, apeland get automatizat
19 persoana.varsta = 30  # Setare directa, apeland set automatizat
20 print(persoana.varsta)
```

La rularea exemplului 7.5 efectul afisat în consola este următorul:

```
25
30
```

În exemplul 7.4, folosim metode get și set tradiționale pentru accesarea și modificarea atributului privat `__varsta`. În exemplul 7.5 , folosim decoratorii `@property` și `@varsta.setter` pentru a crea get și set într-un mod mai simplu și mai natural.

7.4 Decoratorii de proprietate în Python

În Python, decoratorii de proprietate (sau `property decorators`) sunt o caracteristică puternică a limbajului, care permite definirea unor metode *get* și *set* într-un mod mai simplu și elegant. Acestea sunt esențiale pentru controlul accesului la atributele unei clase, oferind în același timp o interfață publică curată și ușor de utilizat.

Decoratorii de proprietate permit adăugarea de logică specifică atunci când o proprietate este accesată sau modificată, fără a necesita apelarea explicită a unor metode speciale de tip *get* sau *set*. Ele promovează încapsularea, validarea datelor și flexibilitatea în modificarea comportamentului unei clase, păstrând în același timp o interfață simplificată pentru utilizatori.

Decoratorii de proprietate în Python folosesc trei decoratori principali:

- `@property`: este folosit pentru a marca o metodă ca *get*, permițând accesul la un atribut prin apelarea metodei ca și cum ar fi o variabilă simplă.
- `@setter`: este utilizat pentru a defini o metodă *set* care permite setarea valorilor pentru acea proprietate.
- `@deleter`: este un decorator mai puțin folosit, care permite definirea unei metode pentru a șterge valoarea unei proprietăți.

Să vedem cum funcționează decoratorii de proprietate printr-un exemplu practic:

Exemplu 7.6 Exemplu, în limbaj Python, de utilizare a get și set, folosind decoratorii de proprietate.

```
1  class ClasaMea:
2      def __init__(self, varsta):
3          self.__varsta = varsta  # Atribut privat
4
5      @property
6      def varsta(self):
7          """Get pentru varsta"""
8          return self.__varsta
9
10     @varsta.setter
11     def varsta(self, varsta):
12         """Set pentru varsta cu validare"""
13         if varsta > 0:
14             self.__varsta = varsta
15         else:
16             raise ValueError("Varsta trebuie sa fie pozitiva!")
17
18     @varsta.deleter
19     def varsta(self):
20         """Deleter pentru varsta"""
21         del self.__varsta
```

În acest exemplu:

- atributul privat `__varsta` este protejat de accesul direct.
- get decorat cu `@property` permite accesarea valorii variabilei `varsta` ca și cum ar fi un atribut simplu.
- set decorat cu `@varsta.setter` permite setarea valorii, adăugând validare.

În Python, principalele avantaje ale utilizării decoratoriilor de proprietate sunt:

- `simplicitate`: accesul la atribute este făcut într-un mod natural și simplu, fără metode dedicate get și set.
- `controlul accesului`: permite controlul asupra citirii și scrierii valorilor, similar cu get și set tradiționali.
- `cod mai clar` : utilizarea proprietăților face codul mai curat și mai ușor de citit.

Omolog, dezavantajele ale utilizării decoratoriilor de proprietate sunt:

- `supraîncarcarea decoratorilor`: pentru clase complexe, utilizarea excesivă a proprietăților și decoratorilor poate face codul mai greu de urmărit.
- `performanță`: similar cu get și set tradiționali, poate introduce o îngreunare a performanței.

8. Conceptul de abstractizare

Conceptul de abstractizare în limbajul C++
Conceptul de abstractizare în limbajul Python
Șabloane (template) în C++
Biblioteci STL în C++

Abstractizarea este unul dintre cei patru piloni fundamentali ai programării orientate pe obiecte (POO), împreună cu încapsularea, moștenirea și polimorfismul. Conceptul de abstractizare constă în ascunderea detaliilor complexe ale unei implementări și expunerea doar a caracteristicilor esențiale care sunt relevante pentru utilizatorul final. Practic, abstractizarea se concentrează pe `ce face` un obiect, mai degrabă decât `cum o face`. Aceasta ajută la reducerea complexității și la crearea unei interfețe clare și simplificate pentru interacțiunea cu obiectele.

În orice domeniu de activitate, abordarea unui proiect porneşte de la construirea unui model cu scopul unei mai bune înţelegeri a problemei de rezolvat. Prin abstractizare se izolează aspectele esenţiale de cele neesenţiale. Pentru o aceeaşi problemă, pot exista mai multe abstractizări, deci mai multe modele. Nu putem vorbi de modele corecte şi modele incorecte, ci de modele adecvate şi modele inadecvate. În abordarea orientată pe obiecte, procesul dezvoltării unui produs software, porneşte de la un model. Aceasta este o abstractizare menită a face posibilă etapa următoare de implementare.

În OOP, abstractizarea este realizată prin intermediul claselor abstracte și al interfețelor. O clasă abstractă definește comportamentul dorit (de exemplu, metodele), dar lasă implementarea specifică a acestor metode la clasele derivate. În mod similar, în Python, abstractizarea este adesea realizată prin clase abstracte și modulele care facilitează lucrul cu interfețele.

Ce oferă Abstractizarea

- **simplificare**: ascunde detalii inutile sau prea complexe ale implementării.
- **modularizare**: permite separarea diferitelor aspecte ale programului.
- **fexibilitate**: ușurează modificarea și extinderea sistemelor software.

Acum, să explorăm abstractizarea în C++ și Python, cu exemple pentru fiecare.

8.1 Conceptul de abstractizare în limbajul C++

În C++, clasele abstracte sunt clase care conțin cel puțin o metodă pur virtuală (o metodă care nu are implementare). Aceste metode sunt doar declarate în clasa abstractă și trebuie implementate de clasele derivate. În continuare vom exemplifica conceptul de abstractizare plecând de la clasa de bază `Formă`.

Exemplu 8.1 Exemplu, în limbaj C++, de clasă abstractă pentru o formă geometrică.

```
#include <iostream>
using namespace std;

class Forma {
public:
    virtual void deseneaza() const = 0; // Metoda pur virtuala
    virtual double aria() const = 0; // Metoda pur virtuala
};

class Cerc : public Forma {
private:
    double raza;
public:
    Cerc(double r) : raza(r) {}
    void deseneaza() const override {
        cout << "Deseneaza un cerc." << endl;
    }
    double aria() const override {
        return 3.14 * raza * raza;
    }
};

class Dreptunghi : public Forma {
private:
    double lungime, latime;
public:
    Dreptunghi(double l, double w) : lungime(l), latime(w) {}
    void deseneaza() const override {
        cout << "Deseneaza un dreptunghi." << endl;
    }
    double aria() const override {
        return lungime * latime;
    }
};
```

```
35
36 int main() {
37     Forma* forma1 = new Cerc(5.0);
38     Forma* forma2 = new Dreptunghi(4.0, 6.0);
39
40     forma1->deseneaza();
41     cout << "Aria: " << forma1->aria() << endl;
42
43     forma2->deseneaza();
44     cout << "Aria: " << forma2->aria() << endl;
45
46     delete forma1;
47     delete forma2;
48     return 0;
49 }
```

La rularea exemplului 8.1 efectul afișat în consolă este următorul:

```
Deseneaza un cerc.
Aria: 78.5
Deseneaza un dreptunghi.
Aria: 24
```

În acest exemplu, clasa `Forma` este o clasă abstractă și definește metodele pur virtuale `deseneaza()` și `aria()`. Clasele `Cerc` și `Dreptunghi` sunt derivate din clasa `Forma` și implementează metodele pur virtuale. Abstractizarea permite manipularea obiectelor de tipuri diferite (`Cerc`, `Dreptunghi`) prin intermediul unei interfețe comune (`Forma`).

Unul dintre cele mai mari avantaje ale abstractizării este că oferă o separare clară între logica de implementare și utilizarea efectivă a obiectelor. Programatorii pot folosi obiectele fără să fie nevoiți să înțeleagă detaliile interne de implementare ale acestora. Spre exemplu, într-un sistem de gestionare a vehiculelor, putem crea o clasă abstractă pentru vehicule, iar clasele derivate precum `Masina` și `Bicicleta` vor implementa metodele specifice.

Exemplu 8.2 Exemplu, în limbaj C++, de separare clară între logica de implementare și utilizarea efectivă a obiectelor.

```
1 #include <iostream>
2 using namespace std;
3
4 class Vehicul {
5 public:
6     virtual void porneste() const = 0;
7 };
8
```

```
9  class Masina : public Vehicul {
10 public:
11     void porneste() const override {
12         cout << "Masina porneste." << endl;
13     }
14 };
15
16 class Bicicleta : public Vehicul {
17 public:
18     void porneste() const override {
19         cout << "Bicicleta porneste." << endl;
20     }
21 };
22
23 int main() {
24     Vehicul* v1 = new Masina();
25     Vehicul* v2 = new Bicicleta();
26
27     v1->porneste();
28     v2->porneste();
29
30     delete v1;
31     delete v2;
32     return 0;
33 }
```

La rularea exemplului 8.2 efectul afișat în consolă este următorul:

```
Masina porneste.
Bicicleta porneste.
```

În acest exemplu, logica abstractă a unui vehicul este separată de detaliile implementării specifice fiecărui tip de vehicul (mașină, bicicletă). Acest lucru simplifică gestionarea mai multor tipuri de vehicule într-un program.

În programarea orientată pe obiecte (OOP), conceptul de abstractizare poate fi abordat în mai multe feluri. Deși există un singur concept general de "abstractizare", acesta poate fi implementat și aplicat în mai multe moduri diferite, în funcție de context. Iată câteva tipuri de abstractizare care pot fi identificate în C++:

Abstractizarea prin interfață (eng. interface abstraction)

Aceasta este cea mai comună formă de abstractizare și implică crearea unei interfețe sau clase abstracte care definește doar ce trebuie să facă un obiect, fără a specifica cum face acest lucru. În C++, acest lucru este realizat prin utilizarea de metode virtuale pure.

Exemplu 8.3 Exemplu, în limbaj C++, de abstractizare prin interfață.

```cpp
#include <iostream>
using namespace std;

class Vehicul {
public:
    virtual void porneste() const = 0;  // Interfata abstracta
    virtual void opreste() const = 0;
};

class Masina : public Vehicul {
public:
    void porneste() const override {
        cout << "Masina porneste." << endl;
    }

    void opreste() const override {
        cout << "Masina se opreste." << endl;
    }
};

int main() {
    Vehicul* v1 = new Masina();

    v1->porneste();
    v1->opreste();

    delete v1;
    return 0;
}
```

La rularea exemplului 8.3, programul afișeaza:

```
Masina porneste.
Masina se opreste.
```

În acest exemplu, `Vehicul` este o clasă abstractă care definește două metode virtuale pure, `porneste()` și `opreste()`. Clasa `Masina` implementează aceste metode, oferind comportamente specifice pentru pornire și oprire. Această interfață abstractă permite tratarea tuturor vehiculelor într-un mod uniform, fără a ne concentra pe detaliile interne ale fiecărui vehicul.

Abstractizare prin tipuri generice (eng. template abstraction)

Abstractizarea poate fi realizată prin utilizarea de `template-uri`, care permit definirea unor structuri generice care pot lucra cu diferite tipuri de date. Acest tip de abstractizare este comun în C++ și permite dezvoltatorilor să scrie cod reutilizabil care poate funcționa cu mai multe tipuri de date.

Exemplu 8.4 Exemplu, în limbaj C++, de abstractizare prin tipuri generice.

```
#include <iostream>
#include <vector>

template <typename T>
class Container {
public:
    virtual void adauga(const T& element) = 0;
    virtual T scoate() = 0;
};

template <typename T>
class Stiva : public Container<T> {
private:
    std::vector<T> elemente;
public:
    void adauga(const T& element) override {
        elemente.push_back(element);
    }

    T scoate() override {
        if (elemente.empty()) {
            throw std::out_of_range("Stiva este goala!");
        }
        T temp = elemente.back();
        elemente.pop_back();
        return temp;
    }
};

int main() {
    Stiva<int> stivaInt;
    stivaInt.adauga(10);
    stivaInt.adauga(20);
    stivaInt.adauga(30);
```

```
35
36      std::cout << "Element scos: " << stivaInt.scoate() << std::
            ↪ endl;
37      std::cout << "Element scos: " << stivaInt.scoate() << std::
            ↪ endl;
38      std::cout << "Element scos: " << stivaInt.scoate() << std::
            ↪ endl;
39
40      try {
41          std::cout << "Element scos: " << stivaInt.scoate() << std
                ↪ ::endl;
42      } catch (const std::out_of_range& e) {
43          std::cerr << "Eroare: " << e.what() << std::endl;
44      }
45
46      return 0;
47  }
```

La rularea exemplului 8.4, programul afișează:

```
Element scos: 30
Element scos: 20
Element scos: 10
Element scos: Eroare: Stiva este goala!
```

Acest exemplu demonstrează utilizarea de template-uri pentru a crea o clasă generică, `Container`, care poate lucra cu orice tip de date. Clasa derivată `Stiva` implementează metodele `adauga()` și `scoate()` pentru a adăuga și a elimina elemente dintr-o structură de stivă (LIFO - Last In, First Out). Abstractizarea template-ului permite reutilizarea codului pentru diverse tipuri de date, precum `int`, `float`, `std::string`, etc.

Abstractizarea funcțională (eng. functional abstraction)

Abstractizarea funcțională se referă la ideea de a defini funcții care ascund detaliile implementării interne și expun doar o interfață pentru utilizatori. Utilizatorii funcției nu trebuie să cunoască modul în care aceasta este implementată, ci doar cum să o utilizeze. Aceasta se aplică în mod special în programarea procedurală, dar este folosită și în OOP pentru a ascunde detaliile interne ale funcțiilor sau metodelor.

Exemplu 8.5 Exemplu, în limbaj C++, de abstractizare funcțională.

```
1  #include <iostream>
2
3  // Definim o interfata abstracta (o clasa cu metode virtuale pure)
4  class Operatie {
```

```
5  public:
6      virtual double calculeaza(double a, double b) = 0; // Metoda
         ↪ pur virtuala = abstractizare functionala
7      virtual ~Operatie() {} // Destructor virtual pentru siguranta
8  };
9
10 // Implementari diferite ale aceleiasi functii abstracte:
11 class Adunare : public Operatie {
12 public:
13     double calculeaza(double a, double b) override {
14         return a + b;
15     }
16 };
17
18 class Scadere : public Operatie {
19 public:
20     double calculeaza(double a, double b) override {
21         return a - b;
22     }
23 };
24
25 class Inmultire : public Operatie {
26 public:
27     double calculeaza(double a, double b) override {
28         return a * b;
29     }
30 };
31
32 class Impartire : public Operatie {
33 public:
34     double calculeaza(double a, double b) override {
35         if (b == 0) throw std::runtime_error("Eroare: Impartire la
             ↪  zero!");
36         return a / b;
37     }
38 };
39
40 int main() {
41     Operatie* operatie; // Pointer catre clasa abstracta
42
43     Adunare adunare;
```

```
44      operatie = &adunare;
45      std::cout << "Adunare: " << operatie->calculeaza(10, 5) << std
            ↪ ::endl;
46
47      Scadere scadere;
48      operatie = &scadere;
49      std::cout << "Scadere: " << operatie->calculeaza(10, 5) << std
            ↪ ::endl;
50
51      Inmultire inmultire;
52      operatie = &inmultire;
53      std::cout << "Inmultire: " << operatie->calculeaza(10, 5) <<
            ↪ std::endl;
54
55      Impartire impartire;
56      operatie = &impartire;
57      try {
58          std::cout << "Impartire: " << operatie->calculeaza(10, 5)
                ↪ << std::endl;
59          std::cout << "Impartire la zero: " << operatie->calculeaza
                ↪ (10, 0) << std::endl;
60      } catch (const std::runtime_error& e) {
61          std::cerr << e.what() << std::endl;
62      }
63
64      return 0;
65  }
```

La rularea exemplului 8.5, programul afișează:

```
Adunare: 15
Scadere: 5
Inmultire: 50
Impartire: 2
Împărțire la zero: Eroare: Împărțire la zero!
```

În acest exemplu, se demonstrează abstractizarea funcțională prin definirea unei interfețe (`operatie`) care expune o metodă abstractă (`calculeaza()`), permițând claselor derivate (`Adunare, Scadere, Inmultire, Impartire`) să implementeze diferite versiuni ale acestei funcționalități, fără ca utilizatorul să fie nevoit să cunoască detaliile specifice fiecărei operații.

Abstractizare prin moștenire (eng. inheritance abstraction)

În moștenire, clasele derivate extind sau modifică comportamentele definite într-o clasă de bază. Prin aceasta, se ascund detaliile implementării din clasa de bază în spatele unei interfețe standardizate care este partajată de toate clasele derivate. Moștenirea în sine este o formă de abstractizare, deoarece oferă posibilitatea de a trata obiecte de tipuri diferite în mod uniform, utilizând interfața comună a clasei de bază.

Exemplu 8.6 Exemplu, în limbaj C++, de abstractizare prin moștenire.

```
1  #include <iostream>
2  using namespace std;
3
4  class Animal {
5  public:
6      virtual void sunet() const = 0; // Metoda pur virtuala ->
       ↪ Clasa devine abstracta
7
8      virtual ~Animal() {} // Destructor virtual pentru siguranta
9  };
10
11 class Caine : public Animal {
12 public:
13     void sunet() const override {
14         cout << "Cainele latra." << endl;
15     }
16 };
17
18 class Pisica : public Animal {
19 public:
20     void sunet() const override {
21         cout << "Pisica toarce." << endl;
22     }
23 };
24
25 int main() {
26     Animal* animal1 = new Caine(); // Polimorfism: folosim pointer
       ↪ la clasa de baza
27     Animal* animal2 = new Pisica();
28
29     animal1->sunet(); // Apelare virtuala: "Cainele latra."
30     animal2->sunet(); // Apelare virtuala: "Pisica toarce."
31
```

```
32      delete animal1; // Curatare memorie
33      delete animal2;
34
35      return 0;
36  }
```

La rularea exemplului 8.6, programul afișeaza:

```
1  Cainele latra.
2  Pisica toarce.
```

În acest exemplu, clasa de bază `Animal` oferă o metodă virtuală `sunet()` care poate fi suprascrisă în clasele derivate, cum ar fi `Caine` și `Pisica`. Detaliile sunetului făcut de fiecare animal sunt ascunse în clasele derivate, iar utilizatorul poate apela metoda `sunet()` pentru orice instanță de `Animal`, fără a se preocupa de tipul specific.

Abstractizare de date (eng. data abstraction)

Abstractizarea de date se concentrează pe ascunderea detaliilor legate de starea internă a obiectelor și oferirea accesului la aceste date prin metode publice (`getteri` și `setteri`). Aceasta ajută la protejarea datelor și la asigurarea controlului asupra modului în care acestea sunt manipulate.

Exemplu 8.7 Exemplu, în limbaj C++, de abstractizare prin funcții set și get.

```
1  #include <iostream>
2  using namespace std;
3
4  class ContBancar {
5  private:
6      double sold;  // Detaliile interne sunt ascunse
7
8  public:
9      ContBancar(double sumaInitiala = 0) : sold(sumaInitiala) {} //
       ↪ Constructor pentru initializare
10
11     void depune(double suma) { // Setter cu validare
12         if (suma > 0) {
13             sold += suma;
14             cout << "Ai depus " << suma << " lei. Sold actual: "
                  ↪ << sold << " lei." << endl;
15         } else {
16             cout << "Suma trebuie sa fie pozitiva!" << endl;
17         }
18     }
```

```
19
20      bool retrage(double suma) { // Setter pentru retragere cu
            ↪ validare
21          if (suma > 0 && suma <= sold) {
22              sold -= suma;
23              cout << "Ai retras " << suma << " lei. Sold actual: "
                    ↪ << sold << " lei." << endl;
24              return true;
25          } else {
26              cout << "Fonduri insuficiente sau suma invalida!" <<
                    ↪ endl;
27              return false;
28          }
29      }
30
31      double getSold() const {  // Getter pentru a accesa soldul
32          return sold;  // Expune doar printr-o metoda controlata
33      }
34  };
35
36  int main() {
37      ContBancar cont(1000); // Creare cont cu sold initial de 1000
            ↪ lei
38
39      cont.depune(500);
40      cont.retrage(300);
41      cont.retrage(1500); // Retragere invalida
42
43      cout << "Sold final: " << cont.getSold() << " lei." << endl;
44
45      return 0;
46  }
```

La rularea exemplului 8.7, programul afișează:

```
Ai depus 500 lei. Sold actual: 1500 lei.
Ai retras 300 lei. Sold actual: 1200 lei.
Fonduri insuficiente sau sumă invalidă!
Sold final: 1200 lei.
```

Clasa `ContBancar` ascunde detaliile legate de sold prin accesul privat la variabila `sold`. Utilizatorul poate interacționa cu această variabilă doar prin metode publice, precum `depune()`

și `getSold()`. Astfel, detaliile interne sunt protejate și se asigură că datele sunt manipulate corect.

Abstractizare de comportament (eng. behavioral abstraction)

În acest tip de abstractizare, comportamentul obiectelor este abstractizat. Aceasta implică definirea unor clase care ascund detaliile modului în care funcționează intern comportamentul acestora, oferind doar o metodă de a interacționa cu comportamentul respectiv.

Exemplu 8.8 Exemplu, în limbaj C++, de abstractizare de comportament.

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  class Dispozitiv {
6  public:
7      virtual void executaComanda(const std::string& comanda) = 0;
       ↪ // Abstractizarea comportamentului
8
9      virtual ~Dispozitiv() {} // Destructor virtual pentru
       ↪ sigurantaa
10 };
11
12 class Televizor : public Dispozitiv {
13 public:
14     void executaComanda(const std::string& comanda) override {
15         cout << "Televizorul executa comanda: " << comanda << endl
           ↪ ;
16     }
17 };
18
19 class Radio : public Dispozitiv {
20 public:
21     void executaComanda(const std::string& comanda) override {
22         cout << "Radio-ul executa comanda: " << comanda << endl;
23     }
24 };
25
26 class AerConditionat : public Dispozitiv {
27 public:
28     void executaComanda(const std::string& comanda) override {
29         cout << "Aerul conditionat executa comanda: " << comanda
```

```
            ↪ << endl;
30      }
31  };
32
33  int main() {
34      vector<Dispozitiv*> dispozitive; // Lista de dispozitive
            ↪ generice
35
36      dispozitive.push_back(new Televizor());
37      dispozitive.push_back(new Radio());
38      dispozitive.push_back(new AerConditionat());
39
40      // Executa aceeasi comanda pentru toate dispozitivele, dar
            ↪ fiecare reactioneaza diferit
41      for (Dispozitiv* d : dispozitive) {
42          d->executaComanda("Porneste");
43      }
44
45      // Curatare memorie
46      for (Dispozitiv* d : dispozitive) {
47          delete d;
48      }
49
50      return 0;
51  }
```

La rularea exemplului 8.8, programul afișează:

```
1  Televizorul executa comanda: Porneste
2  Radio-ul executa comanda: Porneste
3  Aerul conditionat executa comanda: Porneste
```

Clasa abstractă `Dispozitiv` definește un comportament abstract prin metoda `executaComanda()`. Clasele derivate, cum ar fi `Televizor` și `Radio`, implementează acest comportament în funcție de specificul dispozitivului. Această abordare permite tratarea tuturor dispozitivelor într-un mod unificat, fără a cunoaște detaliile specifice fiecărui dispozitiv.

8.2 Conceptul de abstractizare în limbajul Python

În Python, putem folosi modulul `abc` (Abstract Base Class) pentru a crea clase abstracte. În aceste clase, metodele abstracte sunt declarate, dar nu au implementare. Aceste metode trebuie să fie implementate de clasele derivate. În continuare vom exemplifica conceptul de abstractizare în Python pentru o clasa formă geometrică.

Exemplu 8.9 Exemplu, în limbaj Python, de clasă abstractă pentru o formă geometrică.

```
1  from abc import ABC, abstractmethod
2
3  class Forma(ABC):
4      @abstractmethod
5      def deseneaza(self):
6          pass
7
8      @abstractmethod
9      def aria(self):
10         pass
11
12 class Cerc(Forma):
13     def __init__(self, raza):
14         self.raza = raza
15
16     def deseneaza(self):
17         print("Deseneaza un cerc.")
18
19     def aria(self):
20         return 3.14 * self.raza * self.raza
21
22 class Dreptunghi(Forma):
23     def __init__(self, lungime, latime):
24         self.lungime = lungime
25         self.latime = latime
26
27     def deseneaza(self):
28         print("Deseneaza un dreptunghi.")
29
30     def aria(self):
31         return self.lungime * self.latime
32
33 # Exemplu de utilizare
34 forma1 = Cerc(5)
35 forma2 = Dreptunghi(4, 6)
36
37 forma1.deseneaza()
38 print("Aria:", forma1.aria())
39
40 forma2.deseneaza()
```

```
41 print("Aria:", forma2.aria())
```

La rularea exemplului 8.9, programul afișează:

```
1 Deseneaza un cerc.
2 Aria: 78.5
3 Deseneaza un dreptunghi.
4 Aria: 24
```

În acest exemplu, `Forma` este o clasă abstractă, iar metodele `deseneaza()` și `aria()` sunt declarate folosind decoratorul `@abstractmethod`. Clasele `Cerc` și `Dreptunghi` implementează metodele abstracte ale clasei de bază. La fel ca în C++, putem manipula obiecte de tipuri diferite prin intermediul unei interfețe comune.

Abstractizarea în Python poate fi realizată prin diverse metode, cum ar fi clasele abstracte, decoratori, moștenire, generatoare, etc. În continuare sunt prezentate exemple detaliate care ilustrează aceste concepte.

Abstractizarea prin clase abstracte și interfețe

În Python, clasele abstracte sunt definite folosind modulul `abc` (Abstract Base Class). Aceste clase definesc metode abstracte care trebuie implementate de clasele derivate. Acest tip de abstractizare permite definirea unor interfețe clare fără a specifica implementarea.

Exemplu 8.10 Exemplu, în limbaj Python, de abstractizare prin clase abstracte și interfețe.

```
1  from abc import ABC, abstractmethod
2
3  class Vehicul(ABC):
4      @abstractmethod
5      def porneste(self):
6          pass
7
8      @abstractmethod
9      def opreste(self):
10         pass
11
12 class Masina(Vehicul):
13     def porneste(self):
14         print("Masina porneste.")
15
16     def opreste(self):
17         print("Masina se opreste.")
18
19 # Utilizarea claselor abstracte
20 vehicul = Masina()
```

```
vehicul.porneste()  # Afiseaza: Masina porneste.
vehicul.opreste()   # Afiseaza: Masina se opreste.
```

Am definit o clasă abstractă `Vehicul` cu metode abstracte `porneste` și `opreste`. Clasa `Masina` implementează aceste metode. Când instanțiem `Masina`, putem apela metodele, iar acestea vor afișa mesaje specifice.

La rularea exemplului 8.10, programul afișează:

```
Masina porneste.
Masina se opreste.
```

Abstractizarea prin decoratori

Decoratori în Python sunt o formă de abstractizare prin care funcționalități suplimentare pot fi adăugate la funcții sau metode fără a le modifica codul original. Prin decoratori, putem ascunde complexitatea sau detalii inutile.

Exemplu 8.11 Exemplu, în limbaj Python, de abstractizare prin decoratori.

```
def logare(func):
    def wrapper(*args, **kwargs):
        print(f"Executa functia {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

@logare
def salutare(nume):
    print(f"Salut, {nume}!")

# Utilizarea decoratorului
salutare("Andrei")  # Afiseaza: Executand functia salutare
                    #          Salut, Andrei!
```

Decoratorul `logare` adaugă un comportament suplimentar înainte de a executa funcția `salutare`. Afișează un mesaj despre executarea funcției, urmat de salutul propriu-zis.

La rularea exemplului 8.11, programul afișează:

```
Executa functia salutare
Salut, Andrei!
```

Abstractizarea funcțională

Abstractizarea funcțională implică ascunderea detaliilor implementării unei funcții și expunerea doar a comportamentului său dorit. În Python, aceasta poate fi realizată prin funcții de ordin superior (funcții care iau alte funcții ca argumente) sau funcții lambda.

Exemplu 8.12 Exemplu, în limbaj Python, de abstractizare funcțională.

```
def operatiune_aritmetica(operatie, a, b):
    return operatie(a, b)

adunare = lambda a, b: a + b
scadere = lambda a, b: a - b

# Apelarea functiilor abstractizate
print(operatiune_aritmetica(adunare, 5, 3))  # Afiseaza: 8
print(operatiune_aritmetica(scadere, 5, 3))  # Afiseaza: 2
```

Funcția `operatiune_aritmetica` primește o operație ca argument și o aplică asupra numerelor `a` și `b`. Aici, `adunare` și `scadere` sunt funcții lambda care sunt transmise ca parametri.

La rularea exemplului 8.12, programul afișează:

```
8
2
```

Abstractizarea prin moștenire

Moștenirea este o altă formă de abstractizare în Python. Clasele derivate moștenesc comportamentul din clasele de bază și pot suprascrie sau extinde funcționalitatea. Aceasta permite tratarea obiectelor de tipuri diferite într-un mod unificat.

Exemplu 8.13 Exemplu, în limbaj Python, de abstractizare prin moștenire.

```
class Animal:
    def sunet(self):
        print("Animalul face un sunet.")

class Caine(Animal):
    def sunet(self):
        print("Cainele latra.")

class Pisica(Animal):
    def sunet(self):
        print("Pisica toarce.")

# Instantiere si apelare
animale = [Caine(), Pisica()]
for animal in animale:
    animal.sunet()
```

Clasa de bază `Animal` are metoda `sunet`, care este suprascrisă în clasele `Caine` și `Pisica`. Folosind moștenirea, putem trata toate animalele la fel, dar fiecare își va comporta în mod specific.

La rularea exemplului 8.13, programul afișează:

```
1 Cainele latra.
2 Pisica toarce.
```

Abstractizarea prin generatori și funcții `yield`

În Python, generatoarele ascund detalii interne legate de modul în care este produsă o secvență de valori, expunând doar mecanismul de iterare asupra acestora. Aceasta este o formă de abstractizare, deoarece ascunde complexitatea internă a unui algoritm de generare a datelor.

Exemplu 8.14 Exemplu, în limbaj Python, de abstractizare prin generatori și funcții.

```
1 def generator_numere(max):
2     n = 0
3     while n < max:
4         yield n
5         n += 1
6
7 # Utilizarea generatorului
8 for num in generator_numere(5):
9     print(num)
```

Funcția `generator_numere` utilizează `yield` pentru a produce o secvență de valori. Acest generator ascunde complexitatea iterării și permite accesul secvențial la numere.

La rularea exemplului 8.14, programul afișează:

```
1 0
2 1
3 2
4 3
5 4
```

Abstractizarea de date (încapsularea datelor)

Abstractizarea de date în Python poate fi realizată prin controlul accesului la variabilele de instanță utilizând atribute protejate sau private și metode getter și setter. Aceasta permite ascunderea detaliilor implementării și oferă un control mai bun asupra accesului la date.

Exemplu 8.15 Exemplu, în limbaj Python, de abstractizare prin funcții getter și setter.

```
1 class ContBancar:
```

```
2       def __init__(self, sold_initial):
3           self.__sold = sold_initial  # Atribut privat
4
5       def depune(self, suma):
6           self.__sold += suma
7
8       def obtine_sold(self):
9           return self.__sold
10
11  # Utilizarea incapsularii
12  cont = ContBancar(100)
13  cont.depune(50)
14  print(cont.obtine_sold())  # Afiseaza: 150
```

Clasa `ContBancar` ascunde variabila `__sold` prin acces privat. Aceasta poate fi manipulată doar prin metodele publice `depune` și `obtine_sold`, protejând astfel starea internă a obiectului.

La rularea exemplului 8.15, programul afișează:

```
1  150
```

Abstractizarea prin decoratori built-in (proprietăți)

Python oferă decoratori `@property`, care sunt o formă de abstractizare prin care se poate controla accesul la atributele unei clase folosind metode getter și setter, oferind astfel o interfață mai simplificată și mai clară pentru utilizator.

Exemplu 8.16 Exemplu, în limbaj Python, de abstractizare prin decoratori.

```
1   class ContBancar:
2       def __init__(self, sold_initial):
3           self._sold = sold_initial
4
5       @property
6       def sold(self):
7           return self._sold
8
9       @sold.setter
10      def sold(self, suma):
11          if suma >= 0:
12              self._sold = suma
13
14  # Utilizarea decoratorului @property
15  cont = ContBancar(100)
```

```
16 print(cont.sold)  # Afiseaza: 100
17 cont.sold = 200
18 print(cont.sold)  # Afiseaza: 200
```

Folosind decoratori built-in `@property`, `sold` devine un atribut accesibil și modificabil, dar controlat. Aceasta permite protejarea și controlul asupra accesului la date.

La rularea exemplului 8.16, programul afișează:

```
1 100
2 200
```

Abstractizarea prin framework-uri și biblioteci

Python permite abstractizarea prin utilizarea diverselor framework-uri și biblioteci (de exemplu, Django, Flask, NumPy). Acestea ascund detaliile de implementare ale unor funcționalități complexe și oferă o interfață simplificată pentru dezvoltatori. De exemplu, într-un framework web ca Django, nu trebuie să cunoști toate detaliile legate de HTTP sau SQL pentru a dezvolta aplicații web.

Abstractizarea în cadrul unui framework web simplificat, cum ar fi 'Flask', unde nu trebuie să cunoști detaliile despre HTTP sau gestionarea serverelor web.

Exemplu 8.17 Exemplu, în limbaj Python, de abstractizare prin framework-uri și biblioteci.

```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route('/')
6  def index():
7      return "Bine ati venit la aplicatia mea!"
8
9  if __name__ == "__main__":
10     app.run(debug=True)
```

La rularea exemplului 8.17, programul afișează:

```
1 Pornind serverul Flask...
2 Aplicatia va raspunde la http://localhost:5000/
```

8.3 Șabloane (template) în C++

Folosind cuvântul cheie `template` se pot crea şabloane pentru funcţii şi clase, numite funcţii generice, respectiv clase generice. Într-o funcţie sau clasă generică tipul de date asupra căruia operează acestea este specificat ca un parametru. Aceste funcţii şi clase se pot folosi cu diferite tipuri de date fără a rescrie versiunile specifice fiecărui tip.

8.3.1 Funcţii generice

O funcţie generică defineşte un set de operaţii care vor fi aplicate unor tipuri de date variate. În definirea clasică a funcţiilor, pentru a transpune acelaşi algoritm mai multor tipuri de date, este necesară rescrierea procedurii. Dacă se defineşte o funcţie generică, la execuţia funcţiei, compilatorul generează automat codul corect pentru tipul de date folosit efectiv. În esenţă, se creează o funcţie care se supraîncarcă singură automat.

Pentru crearea unei funcţii generice se foloseşte cuvântul cheie `template`. Sintaxa generală folosită pentru definirea unei funcţii generice este:

```
template < class Tip1, class Tip2, ..., class Tipn >
tip_retur nume_functie (lista_parametri)
{
 // corp funcţie
}
```

Tipurile de date sunt precedate de cuvântul cheie `typename`, ceea ce nu înseamnă însă că pot fi doar tipuri de date declarate cu class, ci pot fi orice tipuri de date. `Tip1`, Tip2,..., `Tip n` sunt nume care ţin locul unor tipuri de date folosite de funcţie. Ele ţin doar loc tipurilor de date pe care le va înlocui automat compilatorul la execuţia funcţiei. La apelul funcţiilor generice se pot folosi orice tipuri de date, predefinite sau definite de utilizator. În exemplul 8.18 este definită o funcţie generică de inversare a valorilor a două variabile. Este definită, de asemenea, clasa complex, obiecte de acest tip fiind folosite la apelul funcţiei de inversare.

Exemplu 8.18 Exemplu, în limbaj C++, ce descrie conceptul de funcţie generică (template) pentru inversarea a două variabile.

```
1  #include <iostream>
2
3  template <class T> void inversare ( T & a, T & b)
4  {
5      T aux;
6      aux = a;
7      a = b;
8      b = aux;
9  }
10
11 class complex
12 {
13     double re, im;
14     public:
15         complex(double r = 0, double i = 0)
16         {
17             re = r; im = i;
```

```
18          }
19      void afisare()
20      {
21          std::cout<<"(re="<<re<<" , im="<<im<<" )";
22      }
23  };
24
25  int main()
26  {
27    // declaratii de obiecte de diferite tipuri
28    int p = 7, q = 9;
29    float r = 1.2, s = -2.5;
30    complex c1( 3.1, -6.2 ), c2( r, s );
31
32    std::cout << "\np = " << p << " , q = " << q;
33    std::cout << "\nr = "<< r <<" , s = "<< s;
34    std::cout<<"\nc1: ";
35    c1.afisare();
36    std::cout <<" , c2: ";
37    c2.afisare();
38
39    inversare( p, q ); // apelul functiei cu parametri de tip int
40    inversare( r, s ); // apelul functiei cu parametri de tip float
41    inversare( c1, c2 ); // apelul functiei cu parametri de tip
        ↪ complex
42
43    std::cout << "\np = " << p << " , q = " << q;
44    std::cout << "\nr = " << r << " , s = " << s;
45    std::cout << "\nc1: ";
46    c1.afisare();
47    std::cout << " , c2: ";
48    c2.afisare();
49    std::cin.get();
50    return 0;
51  }
```

La rularea exemplului 8.18, programul afișează:

```
1  p = 7 , q = 9
2  r = 1.2 , s = -2.5
3  c1: (re=3.1 , im=-6.2 ) , c2: (re=1.2 , im=-2.5 )
4  p = 9 , q = 7
```

```
5 r = -2.5 , s = 1.2
6 c1: (re=1.2 , im=-2.5 ) , c2: (re=3.1 , im=-6.2 )
```

Funcţia generică `inversare()` preia referinţa a două variabile de acelaşi tip, desemnat prin numele `T`. La un apel al funcţiei se transmit două date de tip int, la următorul de tip float, iar la al treilea date de tip complex. Un apel de forma: `inversare( p , s );` va fi semnalat printr-un mesaj de eroare, deoarece variabilele `p` şi `s` sunt de tipuri diferite. În exemplul 8.19 este definită o funcţie generică ce preia ca parametri tablouri cu elemente cu tipuri de date diferite, funcţia realizând conversia între cele două tipuri. Funcţia preia şi un parametru de tip int care reprezintă dimensiunea tablourilor.

Exemplu 8.19 Exemplu, în limbaj C++, ce descrie o funcţie generică ce preia ca parametri tablouri cu elemente cu tipuri de date diferite.

```
1 #include <iostream>
2 #include <math.h>
3
4 template <class Tip1, class Tip2>
5 void conv( Tip1 *t1, Tip2 *t2, int n)
6 {
7     for (int i = 0; i < n ; i++ )
8         *(t1 + i) = (Tip1)( *( t2 + i) );
9 }
10
11 class rational
12 {
13     int p, q;
14     public:
15         rational(int a=0, int b=1)
16         {
17             p = a; q = (b!=0 ? b : 1);
18         }
19
20         rational(double d)
21         {
22             p = d*10; q = 10;
23         }
24
25         void citeste();
26
27         void afisare()
28         {
29             std::cout << '(' << p << '/' << q << ')' ;
```

```
30          }
31
32          operator double ()
33          {
34              return (double)p / q ;
35          }
36  };
37
38  void rational::citeste()
39  {
40      std::cout << "\np = ";
41      std::cin >> p;
42       do {
43      std::cout << "q = ";
44      std::cin >> q;
45              }
46      while(q == 0);
47  }
48
49  int main()
50  {
51      int i = 0;
52    int t_i[5]; // tablou cu elemente int
53    double t_d[5]={2.1 , 3.4 , 5.6 , 7.2 , -3}; // tablou cu
        ↪ elemente double
54    rational t_r[5]; // tablou cu elemente rational
55
56    conv( t_i , t_d , 5 ); // se face conversia t_d -> t_i
57    std::cout << std::endl;
58
59    for(int i=0 ; i<5 ; i++) // afisarea elementelor t_i
60      std::cout << t_i[i] << "  ";
61
62    for(i=0 ; i<5 ; i++)
63       t_r[i].citeste();
64
65    conv( t_d , t_r , 5 ); // se face conversia t_r -> t_d
66    std::cout << std::endl;
67
68    for( i=0 ; i<5 ; i++) // afisarea elementelor t_d
69          std::cout << t_d[i] << "  " ;
```

```
70
71    conv( t_r , t_d , 5 ); // se face conversia t_d -> t_r
72    std::cout << std::endl;
73
74    for(i=0 ; i<5 ; i++) // afisarea elementelor t_r
75          t_r[i].afisare();
76    std::cin.get();
77    return 0;
78 }
```

La rularea exemplului 8.19, programul afișează:

```
1  2   3   5   7   -3
2  p = 1
3  q = 2
4
5  p = 2
6  q = 3
7
8  p = 4
9  q = 5
10
11 p = 6
12 q = 7
13
14 p = 8
15 q = 9
16
17 0.5   0.666667   0.8   0.857143   0.888889
18 (5/10)(6/10)(8/10)(8/10)(8/10)
```

În acest exemplu, funcţia generică `conv()` poate fi apelată şi pentru tablouri de tip raţional datorită faptului că sunt definite conversii de la tipul int la raţional, de la tipul double la tipul rational, prin constructorii clasei şi conversia de la tipul rational la double prin funcţia operator `double()`.

În general, funcţiile generice pot prelua parametri de tipuri definite de utilizator în măsura în care conversiile şi operatorii utilizaţi în definiţia funcţiei sunt supradefinite pentru acele tipuri de date. Dacă o funcţie generică nu acoperă cerinţele pentru un anumit tip de date, ea poate fi supradefinită, dar cu restricţia ca în lista de parametri să se regăsească toţi parametrii din funcţia generică.

Exemplul 8.20. defineşte funcţia generică `void ShellSort(T *a, int st, int fn)` care ordonează crescător elementele unui tablou, tipul lor fiind precizat generic prin nu-

mele `T`. Funcţia implementează algoritmul de sortare `ShellSort`. Ea foloseşte operatorul relaţional `<` şi operatorul de atribuire `=`. Aceşti operatori nu pot fi folosiţi pentru şiruri de caractere construite ca `char*`, de aceea, se face o supradefinire a funcţiei care implementează acelaşi algoritm, `void ShellSort( string *a , int st , int fn )`, în care compararea şirurilor se face cu funcţia `strcmp()`, iar atribuirea cu funcţia `strcpy()`.Se observă că listele de parametri ale acelor definiţii sunt similare.

Exemplu 8.20 Exemplu, în limbaj C++, ce descrie funcţia generică void ShellSort pentru a ordona crescător elementele unui tablou.

```
#include <iostream>
#include <string.h>

typedef char string[20];

template <class T>
void ShellSort(T *a, int st, int fn)
{
    int i, j, h;
    T elem_sort;

    h=1;
    do {
        h = 4*h+1;
    } while (h < fn-st + 1);

    do {
        h /= 4;
        for(i = st+h; i <= fn; i++)
        {
            elem_sort = a[i];
            j = i;
            while ( elem_sort < a[j-h] )
            {
                a[j] = a[j-h];
                j -= h;
                if ( j < st+h ) break;
            }
            a[j]=elem_sort;
        }
    } while (h>1);
}
```

```
33
34 //supradefinirea functiei pentru tablou de siruri de caractere
35
36 void ShellSort(string* a , int st , int fn)
37 {
38     int i, j, h;
39     string elem_sort;
40     h=1;
41     do  { h = 3*h+1 ; } while ( h < fn - st + 1) ;
42     do  {
43         h /= 3;
44         for( i = st + h ; i <= fn ; i++ )
45         {
46             strcpy( elem_sort, a[i] );
47             j = i;
48             while ( strcmp( elem_sort , a[j-h] ) < 0 )
49             {
50                 strcpy( a[j] , a[j - h] );
51                 j -= h;
52                 if ( j < st + h) break;
53             }
54             strcpy( a[j] , elem_sort );
55         }
56
57     } while ( h > 1);
58 }
59
60 int main()
61 {
62     int i = 0;
63     int i_vect[25] = {6, 9, 22, 52, 27, 12, 99, 3, 14, 45, 11, 33,
64                       5, 2, 67, 10, 18, 7, 73, 81, 13, 0, 4, 8, 1
                            ↪ };
65
66     ShellSort( i_vect , 0 , 24 );
67     for(int i = 0 ; i < 25 ; i++)
68     std::cout << i_vect[i] << " ";
69     std::cout << "\n";
70
71     string tab[10] ={ "aaa", "Abc", "bcds", "CFdad", "jkdwhksj",
72             "cmnd", "nksjklads", "jjjdklsa", "jdsjla", "jlkdej"};
```

```
73      ShellSort( tab , 0 , 9 );
74
75      for( i = 0 ; i < 10 ; i++)
76          std::cout << "\n" << tab[i];
77      std::cin.get();
78      return 0;
79  }
```

La rularea exemplului 8.20, programul afişează:

```
1  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 18 22 27 33 45 52 67 73 81 99
2
3  Abc
4  CFdad
5  aaa
6  bcds
7  cmnd
8  jdsjla
9  jjjdklsa
10 jkdwhksj
11 jlkdej
12 nksjklads
```

Dacă se defineşte clasa `string` în care se supradefinesc operatorii `<`, respectiv `=`, sunt supradefiniţi, nu mai este necesară supradefinirea funcţiei, definiţia generică putând să folosească şi astfel de parametri.

8.3.2 Clase generice

Sintaxa folosită pentru furnizarea parametrilor unei clase generice este:

```
template <par1, par2,...>
```

Parametrii pot fi de două categorii: tipuri de date sau constante. Tipurile de date sunt precedate de cuvântul cheie class, ceea ce nu înseamnă însă că pot fi doar tipuri de date declarate cu class, ci pot fi orice tip de date. Parametrii care nu sunt precedaţi de cuvântul class sunt consideraţi automat ca fiind constante. De exemplu:

```
template <class Tip>
template <class Tip_el, long Dim>
template <int Dim1, int Dim2>
```

O clasă template, numită şi clasă generică, permite definirea unui şablon pentru definirea de tipuri de date. Declararea şi definirea unei clase generice este totdeauna precedată de clauza template. Considerăm următorul exemplu de clasă generică, numită Stack, pentru reprezentarea de stive.

```
template <class Type> class Stack;
```

Sintaxa de folosire a clauzei template este similară cu cea folosită pentru definirea funcţiilor generice. Definiţia clasei este similară definirii normale, cu excepţia că, în interiorul definiţiei clasei, sunt utilizaţi parametrii tipuri de date. Definirea clasei Stack este prezentată în exemplul 8.21.

Exemplu 8.21 Exemplu, în limbaj C++, ce descrie clasa generică Stack.

```
1  #include <iostream>
2  using namespace std;
3
4  template <class Type>
5  class Stack {
6  private:
7      Type* stack; // Pointer folosit pentru alocarea tabloului de
       ↪ date
8      int top; // Indexul varfului stivei
9      int maxSize; // Dimensiunea maxima a stivei
10
11 public:
12     Stack(int max) : stack(new Type[max]), top(-1), maxSize(max)
       ↪ {}
13
14     ~Stack() {
15         delete[] stack;
16     }
17
18     void Push(Type val) {
19         if (top < maxSize - 1) {
20             stack[++top] = val;
21         } else {
22             cout << "Stiva este plina!" << endl;
23         }
24     }
25
26     void Pop() {
27         if (top >= 0) {
28             --top;
29         } else {
30             cout << "Stiva este goala!" << endl;
31         }
32     }
```

```
33
34      Type& Top() {
35          if (top >= 0) {
36              return stack[top];
37          } else {
38              throw runtime_error("Stiva este goala!");
39          }
40      }
41
42      friend ostream& operator<<(ostream& os, Stack& s) {
43          os << "Stiva: ";
44          for (int i = 0; i <= s.top; i++) {
45              os << s.stack[i] << " ";
46          }
47          return os;
48      }
49  };
50
51  int main() {
52      Stack<int> stiva(5); // Cream o stiva de intregi cu
            ↪ dimensiunea 5
53
54      stiva.Push(10);
55      stiva.Push(20);
56      stiva.Push(30);
57
58      cout << stiva << endl;
59
60      cout << "Elementul din varf: " << stiva.Top() << endl;
61
62      stiva.Pop();
63      cout << "Dupa Pop: " << stiva << endl;
64
65      return 0;
66  }
```

La rularea exemplului 8.21, programul afișează:

```
1  Stiva: 10 20 30
2  Elementul din varf: 30
3  Dupa Pop: Stiva: 10 20
```

Acest cod demonstrează templetizarea claselor prin implementarea unei stive generice (`Stack<Type>`), care poate stoca și manipula elemente de orice tip (`int`, `double`, `string`, etc.), oferind o interfață flexibilă și reutilizabilă. Clasa ascunde detaliile interne ale gestionării memoriei și expune doar operațiile esențiale (`Push()`, `Pop()`, `Top()`), permițând utilizatorului să interacționeze cu stiva fără a se preocupa de implementarea sa internă, ceea ce reprezintă un exemplu de abstractizare și modularitate în C++. Funcţiile membre sunt definite inline, cu excepţia funcţiei `push()`. Operatorul « este, de asemenea, supradefinit printr-o funcție prietenă pentru a afişa conţinutul stivei.

În afară de definiţia propriu-zisă, referirea la o clasă template trebuie să includă lista de parametric template. De aceea, definiţiile acestor funcţii conţin numele `Stack<Type>` în loc de `Stack`. O clasă template reprezintă o formă generică care în momentul implementării va fi folosită pentru crearea de tipuri concrete. Parametrii tipuri de date pot fi atât tipuri fundamentale, cât şi tipuri definite de programator. De exemplu, se pot declara stive cu elemente de tipuri diferite, cum ar fi int, double, pozitie:

```
1 Stack<int> stiva(5); // stiva cu elemente int
2 Stack<double>stiva2(5); // stiva cu elemente double
3 Stack<pozitie>stiva3(5); // stiva cu elemente pozitie
```

Fiecare dintre aceste instanţieri generează funcţii membre ale clasei corespunzătoare fiecărei declaraţii. Astfel, putem scrie secvenţa:

```
1 stiva.Push(10);
2 stiva.Push(20);
3 stiva.Push(30);
4 std::cout << stiva << '\n';
```

care va produce afişarea:

```
10 20 30
```

Când o clasă sau funcţie non-template referă o clasă template, trebuie să existe declaraţie clară a tipului utilizat. De exemplu:

```
1 class Sample {
2     Stack<int>  intStack; // ok
3     Stack<Type> typeStack; // Eroare! Type nu e definit
4     //...
5 };
```

În schimb declaraţia :

```
1 template <class Type>
2 class Sample {
3     Stack<int>  intStack; // ok
4     Stack<Type> typeStack; // ok
```

```
5       //...
6   };
```

este corectă.

Referirea `Stack<int>` reprezintă un specificator valid pentru un tip de date şi poate fi folosit ca orice alt tip de date. Nu numai tipuri de date pot reprezenta parametri pentru clase template, ci şi constante. Iată o altă definiţie a clasei `Stack`, unde valoarea dimensiunii maxime a stivei este transferată ca parametru template, în loc de a fi specificată ca dată membră a clasei (vezi exemplul 8.22.).

Exemplu 8.22 Exemplu, în limbaj C++, de definiție a clasei Stack, unde valoarea dimensiunii maxime a stivei este transferată ca parametru template.

```
1  #include <iostream>
2  using namespace std;
3
4  template <class Type, int maxSize>
5  class Stack {
6  private:
7      Type* stack;  // Pointer pentru alocarea tabloului de date
8      int top; // Indexul varfului stivei
9
10 public:
11     Stack() : stack(new Type[maxSize]), top(-1) {}
12
13     ~Stack() {
14         delete[] stack;
15     }
16
17     void Push(const Type& val) {
18         if (top < maxSize - 1) {
19             stack[++top] = val;
20         } else {
21             cout << "Stiva este plina!" << endl;
22         }
23     }
24
25     void Pop() {
26         if (top >= 0) {
27             --top;
28         } else {
29             cout << "Stiva este goala!" << endl;
30         }
```

```
31      }
32
33      Type& Top() {
34          if (top >= 0) {
35              return stack[top];
36          } else {
37              throw runtime_error("Stiva este goala!");
38          }
39      }
40
41      void Afisare() {
42          cout << "Stiva: ";
43          for (int i = 0; i <= top; i++) {
44              cout << stack[i] << " ";
45          }
46          cout << endl;
47      }
48  };
49
50  int main() {
51      Stack<int, 5> stiva; // Cream o stiva de intregi cu
        ↪ dimensiunea maxima 5
52
53      stiva.Push(10);
54      stiva.Push(20);
55      stiva.Push(30);
56
57      stiva.Afisare(); // Stiva: 10 20 30
58
59      cout << "Elementul din varf: " << stiva.Top() << endl; // 30
60
61      stiva.Pop();
62      stiva.Afisare(); // Stiva: 10 20
63
64      return 0;
65  }
```

La rularea exemplului 8.22, programul afișează:

```
1  Stiva: 10 20 30
2  Elementul din varf: 30
3  Dupa Pop: Stiva: 10 20
```

In exemplul următor, 8.23, este definită o clasa `vector` template cu doi parametri. Tip reprezintă tipul elementelor vectorului, iar `DIM` dimensiunea sa.

Exemplu 8.23 Exemplu, în limbaj C++, ce definește o clasa `vector` template cu doi parametri.

```
1
2  #include <iostream>
3
4  template <class Tip, int DIM>
5  class vector
6  {
7      Tip tab[DIM];
8      public:
9          void citire();
10         void afisare();
11         Tip& operator[](int);
12         Tip operator ~(); // returneaza valoarea maxima a
               ↪ tabloului
13 };
14
15 template <class Tip, int DIM>
16 void vector<Tip, DIM>::citire()
17 {
18     int i;
19     for(i=0; i<DIM; i++)
20     {
21         std::cout<<"vector["<<i<<"]=";
22         std::cin>>tab[i];
23     }
24 }
25
26 template <class Tip, int DIM>
27 void vector<Tip,DIM>::afisare()
28 {
29     int i;
30     std::cout<<std::endl;
31     for(i=0; i<DIM; i++)
32         std::cout<<tab[i]<<"  ";
33 }
34
35 template <class Tip, int DIM>
36 Tip& vector<Tip,DIM>::operator[](int i)
```

```
37 {
38     if (i<DIM)
39         return tab[i];
40     else
41         return tab[0];
42 }
43
44 template <class Tip, int DIM>
45 Tip vector<Tip,DIM>::operator ~()
46 {
47     int i;
48     Tip max=tab[0];
49     for(i=1; i<DIM ; i++)
50         if (max<tab[i])
51             max=tab[i];
52     return max;
53 }
54
55 int main()
56 {
57     vector< int , 6 > v1; // vector cu 6 elemente int
58     vector< float , 5 > v2; // vector cu 5 elemente float
59
60     v1.citire();
61     v1.afisare();
62
63     std::cout<<std::endl<<v1[3];
64     std::cout<<std::endl<<~v1;
65
66     v2.citire();
67     v2.afisare();
68
69     std::cout<<std::endl<<v2[3];
70     std::cout<<std::endl<<~v2;
71
72     std::cin.get();
73 }
```

La rularea exemplului 8.23, programul afișează:

```
1 vector[0]=3
2 vector[1]=4
```

```
3  vector[2]=5
4  vector[3]=6
5  vector[4]=7
6  vector[5]=8
7
8  3   4   5   6   7   8
9  6
10 8
11
12 vector[0]=1
13 vector[1]=2
14 vector[2]=3
15 vector[3]=4
16 vector[4]=5
17
18 1   2   3   4   5
19 4
20 5
```

Spre deosebire exemplul anterior, următorul exemplu, 8.24 definește o clasă vector template cu un parametru, tipul elementelor vectorului, iar dimensiunea sa este stabilită prin membrul dim.

Exemplu 8.24 Exemplu, în limbaj C++, ce definește o clasă `vector` template cu un parametru, tipul elementelor vectorului, iar dimensiunea sa este stabilită prin membrul dim.

```
1  #include <iostream>
2
3  template <class Tip>
4  class vector
5  {
6      Tip* tab;
7      int dim;
8      public:
9          vector(int=0);
10         ~vector();
11         void cit();
12         void af();
13         Tip& operator[](int);
14         Tip operator ~(); // returneaza valoarea maxima a
               ↪ tabloului
15 };
16
```

```
17 template <class Tip>
18 vector<Tip>::vector(int n)
19 {
20     int i;
21     dim=n;
22     tab=new Tip[dim];
23 }
24
25 template <class Tip>
26 vector<Tip>::~vector()
27 {
28     delete [] tab;
29 }
30
31 template <class Tip>
32 void vector<Tip>::cit()
33 {
34     int i;
35     for(i=0; i<dim; i++)
36     {
37         std::cout<<"vector["<<i<<"]=";
38         std::cin>>tab[i];
39     }
40 }
41
42
43 template <class Tip>
44 void vector<Tip>::af()
45 {
46     int i;
47     std::cout<<std::endl;
48     for(i=0; i<dim; i++)
49         std::cout<<tab[i]<<"  ";
50 }
51
52 template <class Tip>
53 Tip& vector<Tip>::operator[](int i)
54 {
55     if (i<dim)
56         return tab[i];
57     else
```

```
58          return tab[0];
59  }
60
61  template <class Tip>
62  Tip vector<Tip>::operator ~()
63  {
64      int i;
65      Tip max=tab[0];
66      for(i=1;i<dim;i++)
67          if (max<tab[i])
68              max=tab[i];
69      return max;
70  }
71
72  int main()
73  {
74      vector<int> v1(6);
75      vector<float> v2(4);
76      v1.cit();
77      v1.af();
78      std::cout<<std::endl<<v1[3];
79      std::cout<<std::endl<<~v1;
80      v2.cit();
81      v2.af();
82      std::cout<<std::endl<<v2[3];
83      std::cout<<std::endl<<~v2;
84      std::cin.get();
85      return 0;
86  }
```

La rularea exemplului 8.24, programul afișează:

```
1  vector[0]=1
2  vector[1]=2
3  vector[2]=3
4  vector[3]=4
5  vector[4]=5
6  vector[5]=5
7
8  1   2   3   4   5   5
9  4
10 5
```

```
11
12 vector[0]=2
13 vector[1]=3
14 vector[2]=4
15 vector[3]=4
16
17 2  3  4  4
18 4
19 4
```

Acest exemplu defineste o clasă vector template cu un parametru, permițând stocarea și manipularea elementelor de orice tip (`int`, `float`, etc.). Clasa include metode pentru citirea și afișarea elementelor, supraincarcă operatorul `[]` pentru acces prin index și definește operatorul `~` pentru a returna valoarea maximă din vector, oferind astfel o soluție generică și reutilizabilă pentru gestionarea tablourilor dinamice în C++.

8.4 Bibliotecii STL în C++

`Standard Template Library (STL)` reprezintă o colecție fundamentală și robustă de biblioteci din limbajul C++, concepută pentru a facilita dezvoltarea rapidă și eficientă de software. STL pune la dispoziția programatorului structuri de date și algoritmi standardizați, care sunt extrem de utili și eficienți. STL este organizată în jurul a trei componente majore:

- `containere` - structuri de date care permit stocarea eficientă a obiectelor.
- `iteratori` - permit navigarea în cadrul containerelor.
- `algoritmi` - funcții generice care operează asupra datelor din containere.

Containere STL în C++

Containerele pot fi clasificate în trei mari categorii:

- `containere secvențiale` - organizează datele într-o secvență liniară. Exemple: `vector`, `list`, `deque`.
- `containere asociative` - organizează datele după o cheie pentru acces rapid. Exemple: `set`, `map`, `multiset`, `multimap`.
- `containere adaptatoare` - adaptează funcționalitatea containerelor secvențiale pentru comportament specific. Exemple: `stack`, `queue`, `priority_queue`.

Următorul exemplu prezintă modul de utilizare a container-ului `map`:

```
1 #include <string>
2 #include <map>
3 #include <iostream>
4
5 int main() {
6 std::map<std::string, int> varste;
7 varste["Ana"] = 25;
```

```
8  varste["Ion"] = 30;
9  varste["Maria"] = 22;
10
11 for(const auto& persoana : varste)
12     std::cout << persoana.first << " are " << persoana.second << "
       ↪  ani.\n";
13
14 return 0;
15 }
```

La rularea exemplului de mai sus, programul afișează:

```
1 Ana are 25 ani.
2 Ion are 30 ani.
3 Maria are 22 ani.
```

Iteratori în STL

Iteratorii permit parcurgerea containerelor STL. Aceștia pot fi:

- `iteratori simpli` - pentru parcurgere simplă.
- `iteratori bidirecționali` - pot parcurge în ambele sensuri (utilizați în containere precum `list`).
- `iteratori cu acces aleator` - permit acces direct la orice element (ex.: `vector`).

Următorul exemplu prezintă modul de utilizare a iteratorului `list`:

```
1  #include <iostream>
2  #include <list>
3  #include <iterator> // pentru std::advance, std::next, etc.
4
5  int main() {
6      std::list<int> numere = {10, 20, 30, 40, 50};
7
8      // Parcurgere inainte cu iterator
9      std::cout << "Parcurgere inainte: ";
10     for (std::list<int>::iterator it = numere.begin(); it !=
       ↪ numere.end(); ++it)
11         std::cout << *it << " ";
12     std::cout << std::endl;
13
14     // Acces la al treilea element folosind advance
15     auto it3 = numere.begin();
16     std::advance(it3, 2);
17     std::cout << "Elementul al treilea este: " << *it3 << std::
```

```
        ↪ endl;

    // Parcurgere inversa folosind reverse_iterator
    std::cout << "Parcurgere inversa: ";
    for (auto rit = numere.rbegin(); rit != numere.rend(); ++rit)
        std::cout << *rit << " ";
    std::cout << std::endl;

    return 0;
}
```

La rularea exemplului de mai sus, programul afișează:

```
Parcurgere inainte: 10 20 30 40 50
Elementul al treilea este: 30
Parcurgere inversa: 50 40 30 20 10
```

Algoritmi STL

STL oferă o colecție vastă de algoritmi standardizați pentru operații comune:

- `sortare și căutare` (`sort`, `binary_search`)
- `copiere și mutare` (`copy`, `move`)
- `ștergere și transformare` (`remove`, `transform`)

Următorul exemplu prezintă modul de utilizare a algoritmului `transform`:

```
#include <iostream>
#include <vector>
#include <algorithm> // pentru std::transform

int patrat(int x) { return x * x; }

int main() {
    std::vector<int> numere = {1, 2, 3, 4, 5};
    std::vector<int> rezultate(numere.size());

    std::transform(numere.begin(), numere.end(), rezultate.begin()
        ↪ , patrat);

    for (auto num : rezultate)
        std::cout << num << " ";

    return 0;
}
```

La rularea exemplului de mai sus, programul afișează:

```
1 1 4 9 16 25
```

STL oferă multiple avantaje esențiale pentru dezvoltarea software:

- `eficiență`: Algoritmii sunt extrem de optimizați pentru performanțe maxime.
- `fiabilitate`: Codul STL este riguros testat și validat în numeroase scenarii.
- `mentenabilitate`: Cod clar și concis, ușor de întreținut și extins.
- `reducerea complexității`: Elimină necesitatea implementării manuale a structurilor și algoritmilor comuni.

Astfel, STL devine o resursă indispensabilă pentru programarea eficientă și profesionistă în C++.

9. Concepte de design patterns

Introducere în design patterns
Patternuri pentru creare (singleton, factory)
Patternuri structurale (adapter, decorator)
Patternuri de comportament (observer, strategy)

9.1 Introducere în design patterns

Modelele de proiectare (eng. Design Patterns) reprezintă soluții reutilizabile și testate în timp pentru probleme comune în dezvoltarea software. Ele oferă o modalitate de a organiza codul, de a crește lizibilitatea și de a îmbunătăți mentenabilitatea aplicațiilor. Design patterns sunt esențiale în dezvoltarea software, deoarece oferă soluții testate și validate pentru probleme comune de design. Prin utilizarea acestor modele, codul devine mai **reutilizabil**, evitând astfel rescrierea inutilă a unor soluții deja existente și dovedite ca eficiente [8]. Un alt beneficiu major al design patterns este **îmbunătățirea mentenanței și scalabilității** aplicațiilor. O arhitectură bine structurată permite adăugarea de noi funcționalități fără a afecta întregul sistem și face mai ușoară corectarea eventualelor erori. De asemenea, aceste modele contribuie la **standardizarea arhitecturii software**, ceea ce facilitează colaborarea între programatori. Folosind un set comun de principii și structuri, echipele pot înțelege și extinde codul mai rapid, indiferent de cine l-a scris inițial. În plus, aplicarea unor soluții deja analizate și testate ajută la **reducerea erorilor**, prevenind problemele care ar putea apărea în cazul unor implementări improvizate. Astfel, design patterns nu doar că economisesc timp, dar și îmbunătățesc calitatea generală a produsului software. Unul dintre cele mai mari avantaje ale utilizării design patterns este **reducerea timpului de dezvoltare**. Deoarece oferă soluții predefinite pentru probleme comune, aceste modele permit programatorilor să se concentreze mai mult pe logica specifică a aplicației și mai puțin pe reinventarea unor concepte fundamentale. De asemenea, design patterns contribuie la **creșterea modularității** codului. Structurile bine definite fac ca fiecare componentă a sistemului să fie clar separată și ușor de înțeles, ceea ce îmbunătățește organizarea proiectului și facilitează extinderea acestuia. Un alt beneficiu semnificativ este **facilitarea testării și depanării**. Deoarece

modelele de design sunt construite pe principii solide și bine structurate, acestea permit identificarea rapidă a erorilor și îmbunătățesc procesul de testare a aplicației. Nu în ultimul rând, design patterns ajută la **reducerea complexității** codului, oferind metode eficiente de organizare a acestuia. Prin utilizarea unor structuri clare și bine definite, aplicațiile devin mai ușor de întreținut și scalat, ceea ce duce la un produs software mai robust și mai eficient.

9.2 Pattern-uri pentru creare

Patternuri pentru creare sunt utilizate pentru a gestiona procesul de creare a obiectelor într-un mod flexibil și eficient. Principalele patternuri pentru creare sunt:

- **singleton** – asigură că există o singură instanță a unei clase și oferă un punct global de acces la aceasta.
- **factory method** – permite crearea de obiecte fără a specifica direct clasa exactă.
- **abstract factory** – oferă o interfață pentru crearea familiilor de obiecte fără a specifica clasele concrete.
- **builder** – separă construcția unui obiect complex de reprezentarea sa finală.

9.2.1 Pattern-uri Singleton

Asigură că există o singură instanță a unei clase și oferă un punct global de acces la aceasta. Următorul exemplu descrie modalitatea de utilizare a Singleton-urilor în C++.

Exemplu 9.1 Exemplu, în limbaj C++, ce definește implementarea pattern-ului Singleton cu asigurarea existentei unei singure instante a unei clase și accesul global la aceasta.

```
1  #include <iostream>
2  #include <memory>
3
4  class Singleton {
5  private:
6      // Constructor privat pentru a preveni instantierea directa
7      Singleton() { std::cout << "Instanta Singleton creata\n"; }
8
9      // Stergem constructorul de copiere si operatorul de atribuire
10     Singleton(const Singleton&) = delete;
11     Singleton& operator=(const Singleton&) = delete;
12
13 public:
14     // Metoda statica pentru a oferi acces global la instanta
15     static Singleton& getInstance() {
16         static Singleton instance; // Garantat ca este initializat
               ↪ o singura data
17         return instance;
18     }
```

```
19
20      void showMessage() {
21          std::cout << "Salutare din Singleton!\n";
22      }
23  };
24
25  int main() {
26      // Accesam instanta Singleton
27      Singleton& s1 = Singleton::getInstance();
28      s1.showMessage();
29
30      // Incercam sa cream o alta instanta (se refera la aceeasi
            ↪ instanta)
31      Singleton& s2 = Singleton::getInstance();
32      s2.showMessage();
33
34      // Confirmam ca ambele instante sunt aceleasi
35      std::cout << "s1 si s2 sunt aceeasi instanta? " << (&s1 == &s2
            ↪  ? "Da" : "Nu") << "\n";
36
37      return 0;
38  }
```

La rularea exemplului 9.1, efectul afișat în consolă este următorul:

```
Instanta Singleton creata
Salutare din Singleton!
Salutare din Singleton!
s1 si s2 sunt aceeasi instanta? Da
```

Pattern-ul **Singleton** este un design pattern care garantează că o clasă are o singură instanță și oferă un punct global de acces la aceasta. În exemplul de mai sus, clasa `Singleton` implementează acest pattern printr-un constructor privat, astfel încât instanțierea directă să fie prevenită. De asemenea, constructorul de copiere și operatorul de atribuire sunt șterse, pentru a preveni copierea instanței.

Metoda `getInstance()` returnează o referință la instanța unică a clasei, care este creată doar la prima apelare a metodei (folosind un obiect static). Aceasta garantează că instanța este creată o singură dată și că toate apelurile ulterioare la `getInstance()` vor returna aceeași instanță.

În funcția `main`, se demonstrează că, deși s-au făcut două apeluri la `getInstance()`, ambele variabile `s1` și `s2` se referă la aceeași instanță, deoarece se verifică dacă adresele de memorie ale acestora sunt identice.

Astfel, acest exemplu ilustrează principiul **Singleton**, asigurându-se că există o singură instanță a clasei `Singleton`.

Echivalent, în linbajul Python, utilizarea pattern-ului Singleton este următoarul:

Exemplu 9.2 Exemplu, în limbaj Python, ce definește implementarea pattern-ului Singleton cu asigurarea existentei unei singure instante a unei clase și accesul global la aceasta.

```
1  class Singleton:
2      _instance = None  # Variabila de clasa pentru stocarea
       ↪ instantei unice
3
4      def __new__(cls):
5          if cls._instance is None:
6              cls._instance = super(Singleton, cls).__new__(cls)
7              print("Instanta Singleton creata")
8          return cls._instance
9
10     def show_message(self):
11         print("Salutare din Singleton!")
12
13 # Exemplu de utilizare
14 if __name__ == "__main__":
15     # Accesam instanta Singleton
16     s1 = Singleton()
17     s1.show_message()
18
19     # Incercam sa cream o alta instanta (se refera la aceeasi
       ↪ instanta)
20     s2 = Singleton()
21     s2.show_message()
22
23     # Confirmam ca ambele instante sunt aceleasi
24     print(f"s1 si s2 sunt aceeasi instanta? {'Da' if s1 is s2 else
       ↪  'Nu'}")
```

9.2.2 Pattern-ul factory method

Factory Method este un design pattern care permite crearea de obiecte fără a specifica direct clasa exactă a acestora. În loc să instanțieze un obiect direct în codul client, o metodă fabrica (factory method) este utilizată pentru a crea obiectul dorit. Aceasta permite decuplarea procesului de creare a obiectelor de restul aplicației și adaugă flexibilitate pentru a schimba tipul de obiect creat fără a modifica codul client.

Următorul exemplu descrie modalitatea de utilizare a Factory Method-urilor în C++14.

Exemplu 9.3 Exemplu, în limbaj C++, ce definește implementarea pattern-ului Factory Method.

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

// Interfata comuna pentru produsele create
class Product {
public:
    virtual void operation() const = 0;  // metoda virtuala pura
    virtual ~Product() = default;
};

// Un tip concret de produs
class ConcreteProductA : public Product {
public:
    void operation() const override {
        std::cout << "Operation of ConcreteProductA\n";
    }
};

// Un alt tip concret de produs
class ConcreteProductB : public Product {
public:
    void operation() const override {
        std::cout << "Operation of ConcreteProductB\n";
    }
};

// Clasa creator, care va contine metoda factory
class Creator {
public:
    virtual std::unique_ptr<Product> createProduct() const = 0;
        ↪ // Factory Method
    virtual ~Creator() = default;
};

// Un creator concret care creeaza ConcreteProductA
class ConcreteCreatorA : public Creator {
public:
    std::unique_ptr<Product> createProduct() const override {
        return std::make_unique<ConcreteProductA>();
    }
```

```
40 };
41
42 // Un alt creator concret care creeaza ConcreteProductB
43 class ConcreteCreatorB : public Creator {
44 public:
45     std::unique_ptr<Product> createProduct() const override {
46         return std::make_unique<ConcreteProductB>();
47     }
48 };
49
50 // Exemplu de utilizare
51 int main() {
52     std::unique_ptr<Creator> creatorA = std::make_unique<
        ↪ ConcreteCreatorA>();
53     std::unique_ptr<Product> productA = creatorA->createProduct();
54     productA->operation();
55
56     std::unique_ptr<Creator> creatorB = std::make_unique<
        ↪ ConcreteCreatorB>();
57     std::unique_ptr<Product> productB = creatorB->createProduct();
58     productB->operation();
59
60     return 0;
61 }
```

La rularea exemplului 9.3, efectul afișat în consolă este următorul:

```
Operation of ConcreteProductA
Operation of ConcreteProductB
```

Pattern-ul **Factory Method** este un design pattern care permite crearea de obiecte fără a specifica direct clasa exactă a obiectului creat. În acest exemplu, avem o ierarhie de clase care demonstrează acest pattern.

Clasa `Product` este o interfață comună pentru produsele care urmează să fie create. Aceasta conține o metodă virtuală pură `operation`, care este implementată de clasele derivate.

`ConcreteProductA` și `ConcreteProductB` sunt două implementări concrete ale clasei `Product`. Fiecare dintre acestea implementează metoda `operation` pentru a furniza comportamente diferite.

Clasa `Creator` este o clasă abstractă care conține metoda `createProduct`, care va fi implementată de clasele concrete. Această metodă este responsabilă pentru crearea produsului, dar fără a specifica tipul concret al acestuia.

`ConcreteCreatorA` și `ConcreteCreatorB` sunt clase concrete care implementează metoda `createProduct` și returnează instanțe de `ConcreteProductA` și `ConcreteProductB`, respectiv.

În funcția `main`, sunt create două obiecte de tip `Creator` (unul pentru `ConcreteProductA` și unul pentru `ConcreteProductB`), iar fiecare creator creează un obiect de tipul corespunzător. Apoi, metoda `operation` este apelată pentru fiecare produs, demonstrând că, deși crearea produselor este făcută printr-o metodă comună (`createProduct`), tipul concret al obiectului este determinat de creatorul specific.

Acest exemplu ilustrează modul în care **Factory Method** permite crearea de obiecte de tipuri diferite fără a face referire directă la clasele concrete, facilitând astfel extinderea și modificarea codului.

Echivalent, in linbajul Python, utilizarea pattern-ului Factory Method este urmatoarul:

Exemplu 9.4 Exemplu, în limbaj Python, ce definește implementarea pattern-ului Singleton cu asigurarea existentei unei singure instanțe a unei clase și accesul global la aceasta.

```
1  from abc import ABC, abstractmethod
2
3  # Interfata comuna pentru produsele create
4  class Product(ABC):
5      @abstractmethod
6      def operation(self):
7          pass
8
9  # Un tip concret de produs
10 class ConcreteProductA(Product):
11     def operation(self):
12         print("Operation of ConcreteProductA")
13
14 # Un alt tip concret de produs
15 class ConcreteProductB(Product):
16     def operation(self):
17         print("Operation of ConcreteProductB")
18
19 # Clasa creator, care va contine metoda factory
20 class Creator(ABC):
21     @abstractmethod
22     def create_product(self):
23         pass
24
25 # Un creator concret care creeaza ConcreteProductA
26 class ConcreteCreatorA(Creator):
```

```
27      def create_product(self):
28          return ConcreteProductA()
29
30  # Un alt creator concret care creeaza ConcreteProductB
31  class ConcreteCreatorB(Creator):
32      def create_product(self):
33          return ConcreteProductB()
34
35  # Exemplu de utilizare
36  if __name__ == "__main__":
37      creatorA = ConcreteCreatorA()
38      productA = creatorA.create_product()
39      productA.operation()
40
41      creatorB = ConcreteCreatorB()
42      productB = creatorB.create_product()
43      productB.operation()
```

9.2.3 Pattern-uri abstract factory

Abstract Factory este un design pattern care oferă o interfață pentru crearea familiilor de obiecte fără a specifica clasele concrete. În loc să creeze obiecte concrete direct, un Abstract Factory va oferi metode care sunt responsabile pentru crearea diferitelor tipuri de obiecte, toate având aceleași interfețe. Aceasta permite schimbarea familiei de obiecte fără a modifica codul client. În continuare se va prezenta câte un exemplu pentru C++ și Python pentru a exemplifica pattern-ul Abstract Factory.

Exemplu 9.5 Exemplu, în limbaj C++, ce definește implementarea pattern-ului Abstract Factory.

```
1  #include <iostream>
2  #include <memory>
3
4  // Interfata pentru produsele familiare
5  class ProductA {
6  public:
7      virtual void operationA() const = 0;
8      virtual ~ProductA() = default;
9  };
10
11 class ProductB {
12 public:
13     virtual void operationB() const = 0;
14     virtual ~ProductB() = default;
```

```
15 };
16
17 // Produse concrete
18 class ConcreteProductA1 : public ProductA {
19 public:
20     void operationA() const override {
21         std::cout << "ConcreteProductA1 operation\n";
22     }
23 };
24
25 class ConcreteProductA2 : public ProductA {
26 public:
27     void operationA() const override {
28         std::cout << "ConcreteProductA2 operation\n";
29     }
30 };
31
32 class ConcreteProductB1 : public ProductB {
33 public:
34     void operationB() const override {
35         std::cout << "ConcreteProductB1 operation\n";
36     }
37 };
38
39 class ConcreteProductB2 : public ProductB {
40 public:
41     void operationB() const override {
42         std::cout << "ConcreteProductB2 operation\n";
43     }
44 };
45
46 // Interfata Abstract Factory
47 class AbstractFactory {
48 public:
49     virtual std::unique_ptr<ProductA> createProductA() const = 0;
50     virtual std::unique_ptr<ProductB> createProductB() const = 0;
51     virtual ~AbstractFactory() = default;
52 };
53
54 // Familie concrete de factori
55 class ConcreteFactory1 : public AbstractFactory {
```

```
56 public:
57     std::unique_ptr<ProductA> createProductA() const override {
58         return std::make_unique<ConcreteProductA1>();
59     }
60
61     std::unique_ptr<ProductB> createProductB() const override {
62         return std::make_unique<ConcreteProductB1>();
63     }
64 };
65
66 class ConcreteFactory2 : public AbstractFactory {
67 public:
68     std::unique_ptr<ProductA> createProductA() const override {
69         return std::make_unique<ConcreteProductA2>();
70     }
71
72     std::unique_ptr<ProductB> createProductB() const override {
73         return std::make_unique<ConcreteProductB2>();
74     }
75 };
76
77 // Exemplu de utilizare
78 int main() {
79     std::unique_ptr<AbstractFactory> factory1 = std::make_unique<
        ↪ ConcreteFactory1>();
80     std::unique_ptr<ProductA> productA1 = factory1->createProductA
        ↪ ();
81     productA1->operationA();
82     std::unique_ptr<ProductB> productB1 = factory1->createProductB
        ↪ ();
83     productB1->operationB();
84
85     std::unique_ptr<AbstractFactory> factory2 = std::make_unique<
        ↪ ConcreteFactory2>();
86     std::unique_ptr<ProductA> productA2 = factory2->createProductA
        ↪ ();
87     productA2->operationA();
88     std::unique_ptr<ProductB> productB2 = factory2->createProductB
        ↪ ();
89     productB2->operationB();
90
```

```
91      return 0;
92  }
```

La rularea exemplului 9.5, efectul afișat în consolă este următorul:

```
ConcreteProductA1 operation
ConcreteProductB1 operation
ConcreteProductA2 operation
ConcreteProductB2 operation
```

În acest exemplu, avem două interfețe de produs (`ProductA` și `ProductB`) care definesc operațiile specifice pentru produsele din fiecare familie.

Clasele `ConcreteProductA1`, `ConcreteProductA2`, `ConcreteProductB1`, și `ConcreteProductB2` sunt implementări concrete ale acestor produse.

Clasa `AbstractFactory` conține metodele `create_productA()` și `create_productB()` care sunt implementate de clasele de factori concreți (`ConcreteFactory1` și `ConcreteFactory2`).

Clienții pot crea obiecte de diferite tipuri din aceleași familii de produse fără a cunoaște clasele concrete. Acest pattern permite schimbarea întregii familii de produse fără a modifica clientul, fiind o alegere excelentă pentru sisteme care trebuie să poată extinde sau modifica ușor tipurile de obiecte create.

Echivalent, în linbajul Python, utilizarea pattern-ului Abstract Factory este următoarul:

Exemplu 9.6 Exemplu, în limbaj Python, ce definește implementarea pattern-ului Abstract Factory.

```
1  from abc import ABC, abstractmethod
2
3  # Interfata pentru produsele familiare
4  class ProductA(ABC):
5      @abstractmethod
6      def operationA(self):
7          pass
8
9  class ProductB(ABC):
10     @abstractmethod
11     def operationB(self):
12         pass
13
14 # Produse concrete
15 class ConcreteProductA1(ProductA):
16     def operationA(self):
17         print("ConcreteProductA1 operation")
18
19 class ConcreteProductA2(ProductA):
20     def operationA(self):
```

```
21          print("ConcreteProductA2 operation")
22
23  class ConcreteProductB1(ProductB):
24      def operationB(self):
25          print("ConcreteProductB1 operation")
26
27  class ConcreteProductB2(ProductB):
28      def operationB(self):
29          print("ConcreteProductB2 operation")
30
31  # Interfata Abstract Factory
32  class AbstractFactory(ABC):
33      @abstractmethod
34      def create_productA(self):
35          pass
36
37      @abstractmethod
38      def create_productB(self):
39          pass
40
41  # Familie concrete de factori
42  class ConcreteFactory1(AbstractFactory):
43      def create_productA(self):
44          return ConcreteProductA1()
45
46      def create_productB(self):
47          return ConcreteProductB1()
48
49  class ConcreteFactory2(AbstractFactory):
50      def create_productA(self):
51          return ConcreteProductA2()
52
53      def create_productB(self):
54          return ConcreteProductB2()
55
56  # Exemplu de utilizare
57  if __name__ == "__main__":
58      factory1 = ConcreteFactory1()
59      productA1 = factory1.create_productA()
60      productA1.operationA()
61      productB1 = factory1.create_productB()
```

```
62      productB1.operationB()
63
64      factory2 = ConcreteFactory2()
65      productA2 = factory2.create_productA()
66      productA2.operationA()
67      productB2 = factory2.create_productB()
68      productB2.operationB()
```

9.2.4 Pattern-ul Builder

Pattern-ul Builder este un design pattern care separă procesul de construire a unui obiect complex de reprezentarea sa finală, permițând astfel crearea diferitelor reprezentări ale obiectului folosind aceeași logică de construire. Pattern-ul Builder este util atunci când avem un obiect complex de construit, care necesită mai multe etape pentru a fi realizat. În loc ca toate aceste etape să fie incluse într-o singură clasă (care ar deveni foarte complexă), pattern-ul Builder propune separarea procesului de construire în mai multe componente. Acesta permite crearea unor reprezentări diferite ale unui obiect, folosind aceleași etape de construire. Astfel, permite construirea unui obiect pas cu pas, fiecare pas fiind responsabil de adăugarea unui aspect al obiectului. Următorul exemplu prezintă modul de utilizare al pattern-urilor Builder.

Exemplu 9.7 Exemplu, în limbaj C++, ce definește implementarea pattern-ului Builder.

```cpp
#include <iostream>
#include <string>
#include <memory>

// Clasa care reprezinta produsul final
class Product {
public:
    void setPartA(const std::string& part) {
        partA = part;
    }

    void setPartB(const std::string& part) {
        partB = part;
    }

    void show() const {
        std::cout << "Part A: " << partA << "\nPart B: " << partB
            ↪ << "\n";
    }

```

```
20 private:
21     std::string partA;
22     std::string partB;
23 };
24
25 // Builder abstract
26 class Builder {
27 public:
28     virtual void buildPartA() = 0;
29     virtual void buildPartB() = 0;
30     virtual std::shared_ptr<Product> getResult() = 0;
31     virtual ~Builder() = default;
32 };
33
34 // ConcreteBuilder care construieste Product-ul
35 class ConcreteBuilder : public Builder {
36 private:
37     std::shared_ptr<Product> product = std::make_shared<Product>()
       ↪ ;
38
39 public:
40     void buildPartA() override {
41         product->setPartA("A1");
42     }
43
44     void buildPartB() override {
45         product->setPartB("B1");
46     }
47
48     std::shared_ptr<Product> getResult() override {
49         return product;
50     }
51 };
52
53 // Directorul care coordoneaza procesul de constructie
54 class Director {
55 private:
56     Builder* builder;
57
58 public:
59     Director(Builder* builder) : builder(builder) {}
```

```
60
61      void construct() {
62          builder->buildPartA();
63          builder->buildPartB();
64      }
65  };
66
67  int main() {
68      ConcreteBuilder builder;
69      Director director(&builder);
70
71      // Construim produsul
72      director.construct();
73
74      // Afisam rezultatul
75      std::shared_ptr<Product> product = builder.getResult();
76      product->show();
77
78      return 0;
79  }
```

La rularea exemplului 9.7, efectul afișat în consolă este următorul:

```
Part A: A1
Part B: B1
```

Clasa `Product` este obiectul final care va fi construit. Clasa `Builder` este o interfață care definește pașii de construire, iar `ConcreteBuilder` este o implementare concretă care construiește efectiv obiectul `Product`. Clasa `Director` este responsabilă pentru a coordona pașii de construire, apelând metodele builder-ului. În `main()`, un `Director` folosește un `ConcreteBuilder` pentru a construi un obiect complex și a-l afisa.

Echivalent, în linbajul Python, utilizarea pattern-ului Builder este urmatoarul:

Exemplu 9.8 Exemplu, în limbaj Python, ce definește implementarea pattern-ului Builder.

```
1  class Product:
2      def __init__(self):
3          self.partA = None
4          self.partB = None
5
6      def set_partA(self, part):
7          self.partA = part
8
9      def set_partB(self, part):
```

```
10          self.partB = part
11
12      def show(self):
13          print(f"Part A: {self.partA}")
14          print(f"Part B: {self.partB}")
15
16
17  class Builder:
18      def build_partA(self):
19          pass
20
21      def build_partB(self):
22          pass
23
24      def get_result(self):
25          pass
26
27
28  class ConcreteBuilder(Builder):
29      def __init__(self):
30          self.product = Product()
31
32      def build_partA(self):
33          self.product.set_partA("A1")
34
35      def build_partB(self):
36          self.product.set_partB("B1")
37
38      def get_result(self):
39          return self.product
40
41
42  class Director:
43      def __init__(self, builder):
44          self.builder = builder
45
46      def construct(self):
47          self.builder.build_partA()
48          self.builder.build_partB()
49
50
```

```
51 if __name__ == "__main__":
52     builder = ConcreteBuilder()
53     director = Director(builder)
54 
55     # Construim produsul
56     director.construct()
57 
58     # Afisam rezultatul
59     product = builder.get_result()
60     product.show()
```

9.3 Pattern-uri structurale

Pattern-urile de structură ajută la organizarea claselor și obiectelor pentru a construi sisteme flexibile și scalabile. Acestea definesc modul în care obiectele sunt compuse pentru a obține noi funcționalități fără a modifica structura internă a componentelor existente.

9.3.1 Pattern-ul Adapter

Pattern-ul Adapter este un design pattern structural care permite ca două interfețe incompatibile să funcționeze împreună. Acesta acționează ca un intermediar între două clase, transformând interfața unei clase existente într-o interfață pe care clienții o pot folosi. Adapterul este util în situații în care avem o clasă existentă, dar interfața acesteia nu este compatibilă cu ceea ce avem nevoie. În loc să modificăm clasa existentă (care poate fi parte dintr-o bibliotecă externă), putem crea un adapter care convertește apelurile către interfața dorită.

Exemplu 9.9 Exemplu, în limbaj C++, ce definește implementarea pattern-ului Adapter.

```
1  #include <iostream>
2  
3  // Clasa existenta (interfata incompatibila)
4  class OldPrinter {
5  public:
6      void legacyPrint(const std::string& text) {
7          std::cout << "Old Printer: " << text << std::endl;
8      }
9  };
10 
11 // Interfata dorita
12 class IPrinter {
13 public:
14     virtual void print(const std::string& text) = 0;
15     virtual ~IPrinter() {}
16 };
```

```
17
18 // Adapter care face legatura intre OldPrinter si IPrinter
19 class PrinterAdapter : public IPrinter {
20 private:
21     OldPrinter oldPrinter;
22 public:
23     void print(const std::string& text) override {
24         oldPrinter.legacyPrint(text);
25     }
26 };
27
28 int main() {
29     IPrinter* printer = new PrinterAdapter();
30     printer->print("Hello, World!");
31     delete printer;
32     return 0;
33 }
```

La rularea exemplului 9.9, efectul afișat în consolă este următorul:

```
Old Printer: Hello, World!
```

În acest exemplu, `OldPrinter` are o metodă `legacyPrint`, dar noi avem nevoie de o interfață `IPrinter` cu metoda `print`. `PrinterAdapter` moștenește `IPrinter` și adaptează apelurile la `OldPrinter`, permițând utilizarea acestuia fără a modifica clasa originală.

În următorul exemplu, este descris, în limbajul Python, pattern-ul Adapter .

Exemplu 9.10 Exemplu, în limbaj Python, ce definește implementarea pattern-ului Adapter.

```
1  # Clasa existenta (interfata incompatibila)
2  class OldPrinter:
3      def legacy_print(self, text):
4          print(f"Old Printer: {text}")
5
6  # Adapter care face legatura intre OldPrinter si interfata dorita
7  class PrinterAdapter:
8      def __init__(self, old_printer):
9          self.old_printer = old_printer
10
11     def print(self, text):
12         self.old_printer.legacy_print(text)
13
14 # Utilizare
15 old_printer = OldPrinter()
```

```
16 printer = PrinterAdapter(old_printer)
17 printer.print("Hello, World!")
```

9.3.2 Patternul Decorator

Pattern-ul Decorator este un design pattern structural care permite adăugarea de funcționalități suplimentare unui obiect în mod dinamic, fără a modifica structura acestuia. Acesta este util atunci când dorim să extindem comportamentul unui obiect fără a modifica clasa de bază. Decoratorul funcționează prin învăluirea (eng.wrapping) unui obiect într-o altă clasă care implementează aceeași interfață și adaugă comportament nou.

Exemplu 9.11 Exemplu, în limbaj C++, ce definește implementarea pattern-ului Decorator.

```
1  #include <iostream>
2
3  // Interfata de baza
4  class IComponent {
5  public:
6      virtual void operation() = 0;
7      virtual ~IComponent() {}
8  };
9
10 // Componenta concreta
11 class ConcreteComponent : public IComponent {
12 public:
13     void operation() override {
14         std::cout << "ConcreteComponent operation" << std::endl;
15     }
16 };
17
18 // Decorator abstract
19 class Decorator : public IComponent {
20 protected:
21     IComponent* component;
22 public:
23     Decorator(IComponent* comp) : component(comp) {}
24     void operation() override {
25         component->operation();
26     }
27 };
28
29 // Decorator concret
30 class ConcreteDecorator : public Decorator {
```

```
31 public:
32     ConcreteDecorator(IComponent* comp) : Decorator(comp) {}
33     void operation() override {
34         Decorator::operation();
35         std::cout << "ConcreteDecorator additional behavior" <<
               ↪ std::endl;
36     }
37 };
38
39 int main() {
40     IComponent* component = new ConcreteComponent();
41     IComponent* decorated = new ConcreteDecorator(component);
42     decorated->operation();
43     delete decorated;
44     delete component;
45     return 0;
46 }
```

La rularea exemplului 9.11, efectul afișat în consolă este următorul:

```
ConcreteComponent operation
ConcreteDecorator additional behavior
```

În acest exemplu, `ConcreteComponent` definește o operație de bază. `ConcreteDecorator` adaugă un comportament suplimentar fără a modifica `ConcreteComponent`, utilizând compoziția și delegarea apelurilor.

În Python, exemplul anterior poate fi rescris după cum urmează.

Exemplu 9.12 Exemplu, în limbaj Python, ce definește implementarea pattern-ului Decorator.

```
1  # Componenta de baza
2  class Component:
3      def operation(self):
4          print("ConcreteComponent operation")
5
6  # Decorator de baza
7  class Decorator:
8      def __init__(self, component):
9          self._component = component
10
11     def operation(self):
12         self._component.operation()
13
14 # Decorator concret
```

```
15 class ConcreteDecorator(Decorator):
16     def operation(self):
17         super().operation()
18         print("ConcreteDecorator additional behavior")
19
20 # Utilizare
21 component = Component()
22 decorated = ConcreteDecorator(component)
23 decorated.operation()
```

9.3.3 Pattern-ul Composite

Pattern-ul Composite este un design pattern structural care permite tratarea unui grup de obiecte ca pe un singur obiect. Acest pattern este util atunci când avem o ierarhie de obiecte și dorim să le manipulăm în mod uniform. În următorul exemplu se va descrie modul lui de lucru.

Exemplu 9.13 Exemplu, în limbaj C++, ce definește implementarea pattern-ului Composite.

```
1  #include <iostream>
2  #include <vector>
3
4  // Interfata componenta comuna
5  class Component {
6  public:
7      virtual void operation() = 0;
8      virtual ~Component() {}
9  };
10
11 // Clasa frunza
12 class Leaf : public Component {
13 public:
14     void operation() override {
15         std::cout << "Leaf operation" << std::endl;
16     }
17 };
18
19 // Clasa compusa care poate contine multiple componente
20 class Composite : public Component {
21 private:
22     std::vector<Component*> children;
23 public:
24     void add(Component* component) {
```

```
25          children.push_back(component);
26      }
27      void operation() override {
28          std::cout << "Composite operation" << std::endl;
29          for (Component* child : children) {
30              child->operation();
31          }
32      }
33  };
34
35  int main() {
36      Composite root;
37      Leaf leaf1, leaf2;
38      root.add(&leaf1);
39      root.add(&leaf2);
40      root.operation();
41      return 0;
42  }
```

La rularea exemplului 9.13, efectul afișat în consolă este următorul:

```
Composite operation
Leaf operation
Leaf operation
```

În Python, exemplul anterior poate fi rescris după cum urmează.

Exemplu 9.14 Exemplu, în limbaj Python, ce definește implementarea pattern-ului Composite.

```
1  class Component:
2      def operation(self):
3          pass
4
5  class Leaf(Component):
6      def operation(self):
7          print("Leaf operation")
8
9  class Composite(Component):
10     def __init__(self):
11         self.children = []
12
13     def add(self, component):
14         self.children.append(component)
15
```

```
16      def operation(self):
17          print("Composite operation")
18          for child in self.children:
19              child.operation()
20
21  # Utilizare
22  root = Composite()
23  leaf1 = Leaf()
24  leaf2 = Leaf()
25  root.add(leaf1)
26  root.add(leaf2)
27  root.operation()
```

9.3.4 Pattern-ul Proxi

Pattern-ul Proxy este un design pattern structural care oferă un substitut sau un înlocuitor pentru un alt obiect. Acesta este utilizat pentru a controla accesul la obiectul real, adăugând un nivel suplimentar de control. Proxy-ul poate fi folosit pentru a implementa funcționalități precum controlul accesului, logarea apelurilor, încărcarea întârziată (lazy loading) sau optimizarea utilizării resurselor.

Exemplu 9.15 Exemplu, în limbaj C++, ce definește implementarea pattern-ului Proxi.

```
1  #include <iostream>
2
3  // Interfata comuna
4  class Subject {
5  public:
6      virtual void request() = 0;
7      virtual ~Subject() {}
8  };
9
10 // Obiectul real
11 class RealSubject : public Subject {
12 public:
13     void request() override {
14         std::cout << "RealSubject handling request." << std::endl;
15     }
16 };
17
18 // Proxy care controleaza accesul la RealSubject
19 class Proxy : public Subject {
20 private:
```

```
21      RealSubject* realSubject;
22  public:
23      Proxy() : realSubject(nullptr) {}
24      ~Proxy() { delete realSubject; }
25      void request() override {
26          if (!realSubject) {
27              realSubject = new RealSubject();
28          }
29          std::cout << "Proxy delegating request to RealSubject." <<
                ↪  std::endl;
30          realSubject->request();
31      }
32  };
33
34  int main() {
35      Proxy proxy;
36      proxy.request();
37      return 0;
38  }
```

La rularea exemplului 9.15, efectul afișat în consolă este următorul:

```
Proxy delegating request to RealSubject.
RealSubject handling request.
```

În acest exemplu, `Proxy` controlează instanțierea și accesul la `RealSubject`. Obiectul real este creat doar atunci când este necesar, economisind astfel resurse. În Python, exemplul anterior poate fi rescris după cum urmează.

Exemplu 9.16 Exemplu, în limbaj Python, ce definește implementarea pattern-ului Proxi.

```
1  # Interfata comuna
2  class Subject:
3      def request(self):
4          pass
5
6  # Obiectul real
7  class RealSubject(Subject):
8      def request(self):
9          print("RealSubject handling request.")
10
11 # Proxy care controleaza accesul la RealSubject
12 class Proxy(Subject):
13     def __init__(self):
```

```
14          self._real_subject = None
15
16      def request(self):
17          if self._real_subject is None:
18              self._real_subject = RealSubject()
19          print("Proxy delegating request to RealSubject.")
20          self._real_subject.request()
21
22  # Utilizare
23  proxy = Proxy()
24  proxy.request()
```

La rularea exemplului 9.16, efectul afișat în consolă este următorul:

```
Proxy delegating request to RealSubject.
RealSubject handling request.
```

În exemplul Python, `Proxy` amână instanțierea obiectului `RealSubject` până când este necesar, oferind astfel un control mai bun asupra utilizării resurselor.

9.4 Pattern-uri comportamentale

Pattern-urile comportamentale (eng. Behavioral Patterns) se concentrează pe modul în care obiectele interacționează între ele și își transferă responsabilitățile. Aceste modele sunt utile pentru a defini comunicarea eficientă și flexibilă între obiecte, evitând dependințele rigide și promovând reutilizarea codului.

Aceste pattern-uri pot fi clasificate în două mari categorii:

- pattern-uri de delegare a responsabilității – definesc modului în care obiectele își transferă responsabilitățile pentru a îmbunătăți modularitatea.
- pattern-uri de comunicare între obiecte – stabilesc mecanisme eficiente pentru schimbul de informații și colaborarea între obiecte.

9.4.1 Pattern-ul Observer

Pattern-ul Observer este un pattern comportamental care definește o relație de tip *publish-subscribe* între obiecte. Acesta permite unui obiect (*Subject*) să își notifice automat mai multe obiecte dependente (*Observers*) despre schimbările sale de stare, fără a le lega direct între ele. Astfel, obiectele *Observers* pot reacționa la modificările stării obiectului *Subject* fără a avea o legătură strânsă cu acesta. Pattern-urile Observer permit unei aplicații să fie flexibilă și extensibilă, reduce dependențele între obiecte și permite adăugarea de noi tipuri de observatori fără a modifica clasa *Subject*.

Exemplu 9.17 Exemplu, în limbaj C++, ce definește implementarea pattern-ului Observer.

```
1  #include <iostream>
2  #include <vector>
```

```cpp
#include <string>
#include <algorithm>  // For std::remove

// Clasa Observer
class Observer {
public:
    virtual void update(const std::string& message) = 0;
};

// Clasa Subject
class Subject {
private:
    std::vector<Observer*> observers;
public:
    void addObserver(Observer* observer) {
        observers.push_back(observer);
    }

    void removeObserver(Observer* observer) {
        // Foloseste o functie lambda ca sa compare corect pointerii
        observers.erase(std::remove_if(observers.begin(), observers.end(),
            [observer](Observer* o) { return o == observer; }), observers.end());
    }

    void notifyObservers(const std::string& message) {
        for (Observer* observer : observers) {
            observer->update(message);
        }
    }
};

// Clasa concreta Observer
class ConcreteObserver : public Observer {
private:
    std::string name;
public:
    ConcreteObserver(const std::string& name) : name(name) {}

```

```
41      void update(const std::string& message) override {
42          std::cout << name << " received message: " << message <<
                ↪ std::endl;
43      }
44  };
45
46  // Main
47  int main() {
48      Subject subject;
49
50      ConcreteObserver observer1("Observer 1");
51      ConcreteObserver observer2("Observer 2");
52
53      subject.addObserver(&observer1);
54      subject.addObserver(&observer2);
55
56      subject.notifyObservers("Hello, Observers!");
57
58      // Removing observer1
59      subject.removeObserver(&observer1);
60      std::cout << "After removing Observer 1:" << std::endl;
61
62      subject.notifyObservers("Hello, remaining Observers!");
63
64      return 0;
65  }
```

La rularea exemplului 9.17, efectul afișat în consolă este următorul:

```
Observer 1 received message: Hello, Observers!
Observer 2 received message: Hello, Observers!
After removing Observer 1:
Observer 2 received message: Hello, remaining Observers!
```

Clasa `Observer` este o clasă abstractă care definește metoda `update`, metodă ce va fi implementată de clasele concrete. Clasa `Subject` ține o listă de observatori și îi notifică atunci când este necesar. Metodele `addObserver`, `removeObserver` și `notifyObservers` sunt folosite pentru a adăuga, elimina și notifica observatori. În cele din urmă, clasa `ConcreteObserver` este implementarea concretă a clasei `Observer`, care reacționează la notificările primite prin suprascrierea metodei `update`

În Python, exemplul anterior poate fi rescris după cum urmează.

Exemplu 9.18 Exemplu, în limbaj Python, ce definește implementarea pattern-ului Observer.

```python
class Observer:
    def update(self, message):
        pass

class Subject:
    def __init__(self):
        self._observers = []

    def add_observer(self, observer):
        self._observers.append(observer)

    def remove_observer(self, observer):
        self._observers.remove(observer)

    def notify_observers(self, message):
        for observer in self._observers:
            observer.update(message)

class ConcreteObserver(Observer):
    def __init__(self, name):
        self.name = name

    def update(self, message):
        print(f"{self.name} received message: {message}")

# Main
subject = Subject()

observer1 = ConcreteObserver("Observer 1")
observer2 = ConcreteObserver("Observer 2")

subject.add_observer(observer1)
subject.add_observer(observer2)

subject.notify_observers("Hello, Observers!")
```

La rularea exemplului 9.18, efectul afișat în consolă este următorul:

```
Observer 1 received message: Hello, Observers!
Observer 2 received message: Hello, Observers!
```

Clasa *Observer* este definită ca o clasă de bază ce include metoda *update*, metodă care va fi implementată de subclasele concrete. Clasa *Subject* gestionează o listă de observatori și

îi notifică pe fiecare atunci când starea se schimbă. Clasa *ConcreteObserver* implementează metoda *update* pentru a reacționa la notificările primite.

În secțiunea principală, se creează un obiect de tip *Subject* și două obiecte de tip *ConcreteObserver*. Când se apelează metoda *notify_ observers*, fiecare observator primește mesajul.

9.4.2 Pattern-ul Strategy

Pattern-ul Strategy este un pattern comportamental care definește o familie de algoritmi, le plasează într-o clasă separată și le face interschimbabile. Acesta permite ca algoritmii să fie selectați dinamic la runtime, fără a modifica codul clientului care utilizează algoritmii. În loc să fie implementat direct în cadrul unei clase, algoritmul este delegat unui obiect separat, iar comportamentul poate fi schimbat dinamic în funcție de necesități. Pattern-ul Strategy permite schimbarea comportamentului unui obiect la runtime, reduce complexitatea clasei, prin delegarea responsabilității unor algoritmi în clase separate, ofera flexibilitate mai mare în adăugarea sau modificarea algoritmilor fără a afecta clientul.

Exemplu 9.19 Exemplu, în limbaj C++, ce definește implementarea pattern-ului Strategy.

```
1  #include <iostream>
2  #include <vector>
3
4  // Interfata Strategy
5  class Strategy {
6  public:
7      virtual int execute(int a, int b) = 0;
8  };
9
10 // Strategie concreta: Adunare
11 class AddStrategy : public Strategy {
12 public:
13     int execute(int a, int b) override {
14         return a + b;
15     }
16 };
17
18 // Strategie concreta: Scadere
19 class SubtractStrategy : public Strategy {
20 public:
21     int execute(int a, int b) override {
22         return a - b;
23     }
24 };
25
```

```cpp
// Contextul care foloseste o strategie
class Context {
private:
    Strategy* strategy;
public:
    void setStrategy(Strategy* strategy) {
        this->strategy = strategy;
    }

    int executeStrategy(int a, int b) {
        return strategy->execute(a, b);
    }
};

int main() {
    Context context;

    // Utilizarea strategiei de adunare
    AddStrategy add;
    context.setStrategy(&add);
    std::cout << "Result of addition: " << context.executeStrategy
        ↪ (10, 5) << std::endl;

    // Utilizarea strategiei de scadere
    SubtractStrategy subtract;
    context.setStrategy(&subtract);
    std::cout << "Result of subtraction: " << context.
        ↪ executeStrategy(10, 5) << std::endl;

    return 0;
}
```

La rularea exemplului 9.19, efectul afișat în consolă este următorul:

```
Result of addition: 15
Result of subtraction: 5
```

În exemplul C++, clasa *Strategy* definește o interfață comună pentru toate strategiile de calcul, iar *AddStrategy* și *SubtractStrategy* sunt implementări concrete ale acestei interfețe. Clasa *Context* utilizează o referință la obiectul *Strategy* pentru a delega execuția unui algoritm specific. Obiectul *Context* poate schimba strategia în timpul execuției, în funcție de nevoi.

Exemplu 9.20 Exemplu, în limbaj Python, ce definește implementarea pattern-ului Strategy.

```python
from abc import ABC, abstractmethod

# Interfata Strategy
class Strategy(ABC):
    @abstractmethod
    def execute(self, a, b):
        pass

# Strategie concreta: Adunare
class AddStrategy(Strategy):
    def execute(self, a, b):
        return a + b

# Strategie concreta: Scadere
class SubtractStrategy(Strategy):
    def execute(self, a, b):
        return a - b

# Contextul care foloseste o strategie
class Context:
    def __init__(self, strategy: Strategy):
        self._strategy = strategy

    def set_strategy(self, strategy: Strategy):
        self._strategy = strategy

    def execute_strategy(self, a, b):
        return self._strategy.execute(a, b)

# Main
context = Context(AddStrategy())
print(f"Result of addition: {context.execute_strategy(10, 5)}")

context.set_strategy(SubtractStrategy())
print(f"Result of subtraction: {context.execute_strategy(10, 5)}")
```

În exemplul Python, clasa *Strategy* este definită ca o clasă abstractă cu o metodă abstractă *execute*. *AddStrategy* și *SubtractStrategy* sunt implementări concrete ale acestei interfețe. Clasa *Context* conține o referință la obiectul *Strategy* și utilizează acest obiect pentru a delega execuția unui algoritm. La fel ca în exemplul C++, strategia poate fi schimbată în

timpul execuției programului.

9.4.3 Pattern-ul Command

Pattern-ul Command este un pattern comportamental care transformă o cerere (o acțiune) într-un obiect independent. Acest obiect conține toate informațiile necesare pentru a executa comanda, permițând astfel parametrizarea, stocarea și anularea acțiunilor. Prin folosirea acestui pattern, obiectele care emit comenzi sunt decuplate de cele care le execută, ceea ce face codul mai flexibil și extensibil. Pattern-ul Command decuplează emitentul comenzii de executantul ei, permite implementarea funcționalităților de anulare (*undo*) și refacere (*redo*) și suportă programarea macrocomenzilor (executarea mai multor comenzi într-o singură operație). Urmatorul exemplu descrie pattern-ul Command, în limbajul C++.

Exemplu 9.21 Exemplu, în limbaj C++, ce definește implementarea pattern-ului Command.

```
1  #include <iostream>
2  #include <vector>
3  #include <memory>
4
5  // Interfata Command
6  class Command {
7  public:
8      virtual void execute() const = 0;
9  };
10
11 // Clasa Receiver care implementeaza logica reala a comenzii
12 class Receiver {
13 public:
14     void action() const {
15         std::cout << "Receiver: Executing action.\n";
16     }
17 };
18
19 // Comanda concreta care apeleaza metoda Receiver-ului
20 class ConcreteCommand : public Command {
21 private:
22     std::shared_ptr<Receiver> receiver;
23 public:
24     ConcreteCommand(std::shared_ptr<Receiver> recv) : receiver(
       ↪ recv) {}
25
26     void execute() const override {
27         receiver->action();
```

```
28      }
29  };
30
31  // Invoker-ul care stocheaza si executa comenzi
32  class Invoker {
33  private:
34      std::vector<std::shared_ptr<Command>> commands;
35  public:
36      void addCommand(std::shared_ptr<Command> command) {
37          commands.push_back(command);
38      }
39
40      void executeCommands() const {
41          for (const auto& command : commands) {
42              command->execute();
43          }
44      }
45  };
46
47  // Main
48  int main() {
49      std::shared_ptr<Receiver> receiver = std::make_shared<Receiver
            ↪ >();
50      std::shared_ptr<Command> command = std::make_shared<
            ↪ ConcreteCommand>(receiver);
51
52      Invoker invoker;
53      invoker.addCommand(command);
54
55      invoker.executeCommands();
56
57      return 0;
58  }
```

La rularea exemplului 9.21, efectul afișat în consolă este următorul:

```
Receiver: Executing action.
```

Pattern-ul *Command* presupune definirea unei interfețe abstracte care conține metoda `execute`, metodă pe care toate comenzile trebuie să o implementeze. Clasa *ConcreteCommand* reprezintă o implementare concretă a unei comenzi, având rolul de a apela metoda `action` a obiectului `Receiver`. Clasa *Receiver* este responsabilă pentru implementarea logicii reale a acțiunii solicitate. Obiectul *Invoker* gestionează comenzile și le execută atunci

când este necesar. În funcția `main`, se creează un obiect `Receiver`, care este utilizat de o comandă concretă. Invoker-ul stochează această comandă și o execută la cerere.

În Python, exemplul anterior poate fi rescris după cum urmează.

Exemplu 9.22 Exemplu, în limbaj Python, ce definește implementarea pattern-ului Command.

```
1  from abc import ABC, abstractmethod
2
3  # Interfata Command
4  class Command(ABC):
5      @abstractmethod
6      def execute(self):
7          pass
8
9  # Clasa Receiver care implementeaza logica reala a comenzii
10 class Receiver:
11     def action(self):
12         print("Receiver: Executing action.")
13
14 # Comanda concreta care apeleaza metoda Receiver-ului
15 class ConcreteCommand(Command):
16     def __init__(self, receiver: Receiver):
17         self.receiver = receiver
18
19     def execute(self):
20         self.receiver.action()
21
22 # Invoker-ul care stocheaza si executa comenzi
23 class Invoker:
24     def __init__(self):
25         self.commands = []
26
27     def add_command(self, command: Command):
28         self.commands.append(command)
29
30     def execute_commands(self):
31         for command in self.commands:
32             command.execute()
33
34 # Main
35 if __name__ == "__main__":
36     receiver = Receiver()
```

```
37      command = ConcreteCommand(receiver)
38
39      invoker = Invoker()
40      invoker.add_command(command)
41
42      invoker.execute_commands()
```

Clasa *Command* este o clasă abstractă care definește metoda `execute`, metodă ce trebuie implementată de toate comenzile concrete. Clasa *ConcreteCommand* reprezintă o implementare concretă a unei comenzi, având rolul de a apela metoda `action` a obiectului `Receiver`. Clasa *Receiver* conține logica reală a acțiunii care trebuie executată. Obiectul *Invoker* are rolul de a stoca comenzile și de a le executa atunci când este necesar. În secțiunea principală a programului, obiectul `Receiver` este utilizat prin intermediul unei comenzi concrete, care este stocată și executată de către `Invoker`.

9.4.4 Pattern-ul Mediator

Pattern-ul Mediator este un pattern comportamental care reduce dependențele dintre obiecte, facilitând comunicarea acestora printr-un obiect central numit *mediator*. În loc ca obiectele să comunice direct între ele, ele trimit mesaje către mediator, care le redirecționează către destinatarii potriviți. Acest model ajută la reducerea complexității și a dependințelor între componente. Pattern-ul Mediator reduce dependințele dintre obiecte, evitând conexiuni directe între ele, simplifică modificarea comportamentului sistemului fără a afecta multe clase, îmbunătățește scalabilitatea și întreținerea codului.

Urmatorul exemplu descrie pattern-ul Mediator, în limbajul C++.

Exemplu 9.23 Exemplu, în limbaj C++, ce definește implementarea pattern-ului Mediator.

```
1  #include <iostream>
2  #include <vector>
3  #include <memory>
4
5  // Forward declaration
6  class Colleague;
7
8  // Interfata Mediator
9  class Mediator {
10 public:
11     virtual void sendMessage(const std::string& message, Colleague
       ↪ * sender) = 0;
12 };
13
14 // Clasa Colleague de baza
15 class Colleague {
```

```
16 protected:
17     Mediator* mediator;
18 public:
19     Colleague(Mediator* m) : mediator(m) {}
20     virtual void receiveMessage(const std::string& message) = 0;
21 };
22
23 // Clasa concreta de Mediator
24 class ConcreteMediator : public Mediator {
25 private:
26     std::vector<Colleague*> colleagues;
27 public:
28     void addColleague(Colleague* colleague) {
29         colleagues.push_back(colleague);
30     }
31
32     void sendMessage(const std::string& message, Colleague* sender
       ↪ ) override {
33         for (auto* colleague : colleagues) {
34             if (colleague != sender) {
35                 colleague->receiveMessage(message);
36             }
37         }
38     }
39 };
40
41 // Implementarea concreta a Colleague
42 class ConcreteColleague : public Colleague {
43 private:
44     std::string name;
45 public:
46     ConcreteColleague(Mediator* m, std::string n) : Colleague(m),
       ↪ name(n) {}
47
48     void sendMessage(const std::string& message) {
49         std::cout << name << " sends: " << message << std::endl;
50         mediator->sendMessage(message, this);
51     }
52
53     void receiveMessage(const std::string& message) override {
54         std::cout << name << " receives: " << message << std::endl
```

```
            ↪ ;
55      }
56  };
57
58  // Main
59  int main() {
60      ConcreteMediator mediator;
61
62      ConcreteColleague c1(&mediator, "Alice");
63      ConcreteColleague c2(&mediator, "Bob");
64      ConcreteColleague c3(&mediator, "Charlie");
65
66      mediator.addColleague(&c1);
67      mediator.addColleague(&c2);
68      mediator.addColleague(&c3);
69
70      c1.sendMessage("Hello, everyone!");
71      c2.sendMessage("Hi, Alice!");
72
73      return 0;
74  }
```

La rularea exemplului 9.23, efectul afișat în consolă este următorul:

```
Alice sends: Hello, everyone!
Bob receives: Hello, everyone!
Charlie receives: Hello, everyone!
Bob sends: Hi, Alice!
Alice receives: Hi, Alice!
Charlie receives: Hi, Alice!
```

Clasa *Mediator* este o interfață abstractă care definește metoda `sendMessage`, având rolul de a facilita comunicarea între obiecte. Clasa *ConcreteMediator* reprezintă o implementare concretă a mediatorului, gestionând o listă de colegi și distribuind mesajele primite între aceștia. Clasa *Colleague* servește drept clasă de bază pentru obiectele care comunică prin intermediul mediatorului. Clasa *ConcreteColleague* este o implementare concretă a clasei *Colleague*, permițând trimiterea și primirea de mesaje. În funcția `main`, se creează un obiect mediator și trei colegi care comunică între ei prin intermediul acestuia.

În Python, exemplul anterior poate fi rescris după cum urmează.

Exemplu 9.24 Exemplu, în limbaj Python, ce definește implementarea pattern-ului Mediator.

```
1  from abc import ABC, abstractmethod
2
```

```
3  # Interfata Mediator
4  class Mediator(ABC):
5      @abstractmethod
6      def send_message(self, message, sender):
7          pass
8
9  # Clasa Colleague de baza
10 class Colleague(ABC):
11     def __init__(self, mediator):
12         self.mediator = mediator
13
14     @abstractmethod
15     def receive_message(self, message):
16         pass
17
18 # Clasa concreta de Mediator
19 class ConcreteMediator(Mediator):
20     def __init__(self):
21         self.colleagues = []
22
23     def add_colleague(self, colleague):
24         self.colleagues.append(colleague)
25
26     def send_message(self, message, sender):
27         for colleague in self.colleagues:
28             if colleague != sender:
29                 colleague.receive_message(message)
30
31 # Implementarea concreta a Colleague
32 class ConcreteColleague(Colleague):
33     def __init__(self, mediator, name):
34         super().__init__(mediator)
35         self.name = name
36
37     def send_message(self, message):
38         print(f"{self.name} sends: {message}")
39         self.mediator.send_message(message, self)
40
41     def receive_message(self, message):
42         print(f"{self.name} receives: {message}")
43
```

```
44 # Main
45 if __name__ == "__main__":
46     mediator = ConcreteMediator()
47
48     c1 = ConcreteColleague(mediator, "Alice")
49     c2 = ConcreteColleague(mediator, "Bob")
50     c3 = ConcreteColleague(mediator, "Charlie")
51
52     mediator.add_colleague(c1)
53     mediator.add_colleague(c2)
54     mediator.add_colleague(c3)
55
56     c1.send_message("Hello, everyone!")
57     c2.send_message("Hi, Alice!")
```

La rularea exemplului 9.24, efectul afișat în consolă este următorul:

```
Alice sends: Hello, everyone!
Bob receives: Hello, everyone!
Charlie receives: Hello, everyone!
Bob sends: Hi, Alice!
Alice receives: Hi, Alice!
Charlie receives: Hi, Alice!
```

10. Practici uzuale în programare

Scrierea unui cod curat și ușor de întreținut
Linii directoare pentru convențiile de denumire și consistență semantică
Considerații privind gestionarea memoriei
Tehnici de depanare

10.1 Scrierea unui cod curat si ușor de intreținut

Un cod curat este esențial pentru dezvoltarea de software sustenabil, ușor de înțeles și de modificat. Un cod bine structurat nu doar reduce timpul necesar depanării și întreținerii, ci și îmbunătățește colaborarea între membrii echipei. În acest capitol, vom explora bune practici pentru scrierea unui cod clar, concis și ușor de întreținut în C++ și Python, punând accent pe convențiile de denumire, gestionarea memoriei și tehnicile de depanare.

Pentru a scrie un cod curat, este important să respectăm anumite principii fundamentale care îmbunătățesc lizibilitatea și întreținerea codului. Printre cele mai importante se numără:

- **KISS (Keep It Simple, Stupid)** - codul trebuie să fie simplu și ușor de înțeles. Principiul KISS subliniază importanța menținerii codului simplu și ușor de înțeles. Codul complex este mai greu de întreținut și de depanat.
- **DRY (Don't Repeat Yourself)** - evitarea duplicării codului prin refactorizare. O clasă trebuie să fie deschisă pentru extindere, dar închisă pentru modificare.
- **YAGNI (You Ain't Gonna Need It)** - evitarea adăugării premature a funcționalităților inutile. YAGNI sugerează că nu trebuie să adăugăm funcționalități inutile care nu sunt necesare în prezent.
- **SOLID** - un set de principii pentru o arhitectură robustă. Fiecare clasă trebuie să aibă o singură responsabilitate.

Principiile SOLID sunt printre cele mai cunoscute între programatori și, la rândul lor, sunt formate dintr-o serie de sub-principii după cum urmează:

- **Single Responsibility Principle (SRP)** - fiecare clasă ar trebui să aibă o singură responsabilitate.
- **Open/Closed Principle (OCP)** - entitățile software trebuie să fie deschise pentru extindere, dar închise pentru modificare.

- **Liskov Substitution Principle (LSP)** - obiectele derivate trebuie să poată fi utilizate fără a modifica comportamentul programului. O subclasă trebuie să poată înlocui o superclasă fără a modifica comportamentul programului.
- **Interface Segregation Principle (ISP)** - o interfață nu trebuie să forțeze clienții să implementeze metode inutile. O interfață nu trebuie să forțeze clienții să implementeze metode inutile.
- **Dependency Inversion Principle (DIP)** - modulele de nivel înalt nu trebuie să depindă de cele de nivel jos, ci ambele trebuie să depindă de abstracții. Modulele de nivel înalt nu trebuie să depindă de modulele de nivel jos, ci ambele trebuie să depindă de abstracții.

10.2 Linii directoare pentru convențiile de denumire și consistență semantică

O mare parte din claritatea unui cod derivă din denumirile alese pentru clase, metode, funcții și variabile. Un nume bine ales reduce drastic nevoia de comentarii explicative și permite cititorului să înțeleagă intenția autorului fără să analizeze detaliile implementării.

În ambele limbaje, este recomandat ca numele claselor să fie scrise în format `PascalCase`, adică fiecare cuvânt important să înceapă cu literă mare. De exemplu, o clasă care se ocupă cu procesarea datelor ar putea fi denumită `DataProcessor`, atât în C++, cât și în Python:

```
1 // C++
2 class DataProcessor {
3     ...
4 };
5
6 // Python
7 class DataProcessor:
8     ...
```

În cazul funcțiilor și metodelor, convențiile diferă ușor între cele două limbaje. În C++, este destul de răspândit stilul `CamelCase`, dar mulți adoptă și `snake_case`, mai ales în contexte moderne sau când folosesc frameworkuri. În schimb, Python urmează cu strictețe ghidul PEP8, care recomandă utilizarea stilului `snake_case`. Prin urmare, o funcție care calculează suma totală ar putea arăta astfel:

```
1 // C++
2 double calculeazaTotal(double pret, double rataTaxare);
3
4 // Python
5 def calculeaza_total(prex, rata_taxare):
6     ...
```

Când vine vorba de variabile, evitarea prescurtărilor criptice este esențială. O variabilă numită `a` sau `x1` nu oferă nicio informație despre semnificația ei, în timp ce `pret_total` sau

`numar_utilizatori` fac codul mult mai lizibil. De asemenea, consistența este crucială: dacă într-o parte a proiectului folosești `num_utilizatori`, iar în altă parte `numar_utilizatori`, riști să creezi confuzie.

10.2.1 Structurarea codului: simplitate și separarea responsabilităților

Un cod ușor de întreținut este unul în care fiecare parte are un scop clar și bine delimitat. O funcție ar trebui să facă un singur lucru și să îl facă bine. În momentul în care o funcție începe să se ocupe de mai multe responsabilități – să citească date, să le proceseze, să afișeze rezultate și să trateze erori – devine dificil de testat, de reutilizat și mai ales de înțeles. Acesta este motivul pentru care principiul responsabilității unice (SRP – Single Responsibility Principle) este atât de important în programarea orientată pe obiect.

În C++, acest principiu se reflectă prin crearea unor clase bine delimitate, unde fiecare clasă se ocupă doar de un aspect al aplicației. De exemplu, o clasă `CalculatorTaxe` ar trebui să aibă ca singur scop calcularea taxelor și nu ar trebui să știe nimic despre afișarea rezultatelor sau despre sursa datelor.

```
// C++
class CalculatorTaxe {
public:
    double calculeaza(double pret, double rate) {
        return pret * rate;
    }
};
```

În mod similar, în Python putem avea o clasă minimalistă, concentrată doar pe logica de calcul:

```
class CalculatorRate:
    def calculate(self, price, rate):
        return pret * rate
```

10.2.2 Comentarii, stil și lizibilitate

Comentariile trebuie folosite cu moderație și doar acolo unde este necesar să explicăm *de ce* se face ceva, nu *ce* se face. Dacă numele funcțiilor și variabilelor sunt alese corect, atunci codul va fi auto-explicativ în cea mai mare parte.

Pe lângă comentarii, stilul de scriere contează enorm: indentarea trebuie să fie consistentă (Python o impune prin sintaxă, dar în C++ trebuie respectată prin disciplină), iar blocurile de cod ar trebui separate vizual pentru a ușura parcurgerea logicii. Un cod bine formatat reduce timpul de înțelegere și șansele de a introduce erori la modificări ulterioare.

10.2.3 Organizarea proiectului și separarea logicii de implementare

În proiectele reale, codul crește rapid în dimensiune, iar organizarea lui devine esențială. În C++, este o bună practică să separi declarațiile (în fișiere `.h`) de implementare (în fișiere `.cpp`), astfel încât interfața să fie vizibilă fără a fi necesar să citești detaliile implementării. În Python, organizarea pe module și pachete, fiecare având o responsabilitate clară, ajută la păstrarea codului lizibil și testabil.

De asemenea, includerea unor teste unitare simple este un pas esențial în întreținerea codului. Un cod netestat este un cod fragil, pe care programatorii se tem să-l atingă. Testele nu doar că previn regresiile, dar acționează și ca o formă de documentație vie: ele arată cum ar trebui să fie folosite clasele și metodele.

10.3 Considerații privind gestionarea memoriei

Gestionarea memoriei este unul dintre subiectele fundamentale în programarea de sistem și un aspect care influențează profund designul aplicațiilor, mai ales atunci când vorbim despre programare orientată pe obiecte. Modul în care o aplicație alocă, utilizează și eliberează memoria determină nu doar performanța sa, ci și fiabilitatea și siguranța acesteia. Diferențele între C++ și Python sunt, din acest punct de vedere, semnificative și merită discutate în detaliu.

În continuare se va prezenta cum este tratată memoria în cele două limbaje, ce responsabilități are programatorul și ce rol joacă programarea orientată pe obiecte în gestionarea resurselor.

10.3.1 Memorie gestionată explicit vs. automat

C++ este un limbaj care oferă control aproape total asupra memoriei. Acest lucru îl face extrem de puternic, dar în același timp și periculos, întrucât responsabilitatea pentru alocarea și eliberarea memoriei cade în totalitate pe umerii programatorului. Fiecare apel la `new` trebuie, în mod ideal, să fie însoțit de un `delete` corespunzător. Nerespectarea acestei reguli poate duce la scurgeri de memorie (*memory leaks*) sau, mai grav, la accesări ilegale ale memoriei eliberate.

```
1 // C++
2 ClasaMea* obiect = new ClasaMea();
3 // ... utilizarea obiectului
4 delete obiect; // important!
```

Pe de altă parte, Python utilizează un sistem de gestionare automată a memoriei, bazat pe un mecanism de referințe și un colector de gunoi (*garbage collector*). Deși programatorul nu este nevoit să elibereze manual memoria, este totuși important să înțeleagă cum funcționează acest mecanism și când poate deveni problematic – de exemplu, în cazul referințelor circulare sau când sunt păstrate referințe inutile la obiecte mari.

```
1 # Python
2 obiect = ClasaMea()
```

```
3 # obiectul va fi eliberat automat cand nu mai este referentiat
```

Această diferență fundamentală are un impact direct asupra stilului de programare: în C++, trebuie să fim permanent atenți la ciclul de viață al obiectelor, în timp ce în Python ne putem concentra mai mult pe logică și mai puțin pe gestionarea resurselor de jos.

10.3.2 Destructori, context și RAII

Un concept esențial în C++ este RAII – *Resource Acquisition Is Initialization*. Conform acestuia, resursele (fie că este vorba de memorie, fișiere, mutexuri sau conexiuni de rețea) sunt asociate cu durata de viață a unui obiect. Când obiectul este distrus (de regulă automat, la ieșirea din scope), resursele sunt eliberate automat, prin intermediul destructorului.

```
1  // C++
2  class GestiuneFisier {
3  public:
4      GestiuneFisier(const std::string& cale) {
5          fisier = fopen(cale.c_str(), "r");
6      }
7      ~GestiuneFisier() {
8          if (fisier) fclose(fisier);
9      }
10 private:
11     FILE* fisier;
12 };
```

RAII permite programatorului să evite `delete`-uri sau `fclose`-uri scrise manual în fiecare ramură logică. Este o paradigmă extrem de utilă pentru un cod robust, dar presupune o bună înțelegere a ciclului de viață al obiectelor.

În Python, un model similar este implementat prin context manageri și cuvântul-cheie `with`. Acesta oferă un mod elegant de a gestiona resurse temporare, fără a lăsa programatorului grijile eliberării explicite.

```
1 # Python
2 with open("fisier.txt") as f:
3     continut = f.read()
4 # fisierul este inchis automat la iesirea din bloc
```

Pentru clase personalizate, Python permite implementarea protocoalelor `__enter__` și `__exit__`, permițând crearea de clase care se comportă ca niște contexte, similar cu RAII în C++.

10.3.3 Clase de bază, moștenire și alocare dinamică

În programarea orientată pe obiecte, utilizarea moștenirii și a polimorfismului implică adesea alocarea dinamică a obiectelor. În C++, acest lucru poate duce la probleme dacă

distrugerea obiectelor derivate nu este gestionată corect. Un caz clasic este folosirea destructorilor virtuali:

```cpp
// C++
class Baza {
public:
    virtual ~Baza() {} // important!
};

class Derivata : public Baza {
public:
    ~Derivata() {
        std::cout << "Distrusa\n";
    }
};

Baza* obiect = new Derivata();
delete obiect; // fara destructor virtual, nu se apeleaza ~
    ↪ Derivata!
```

În Python, toate obiectele sunt alocate dinamic și toate clasele derivă implicit din `object`. Distrugerea este determinată de sistemul de referințe, iar metodele `__del__` pot fi folosite pentru a implementa destructori. Totuși, acestea nu sunt recomandate în general, din cauza comportamentului imprevizibil în prezența ciclurilor de referință.

10.3.4 Optimizări și bune practici

Chiar dacă în Python memoria este gestionată automat, programatorul nu este complet scutit de responsabilitate. Păstrarea inutilă a obiectelor în variabile globale, folosirea unor structuri de date ineficiente sau încărcarea excesivă a memoriei prin cache-uri nereglementate pot duce la consum excesiv de memorie și performanțe slabe.

În C++, utilizarea de pointeri inteligenți precum `std::unique_ptr` și `std::shared_ptr` este o practică recomandată. Aceste tipuri moderne de pointeri automatizează eliberarea memoriei fără a sacrifica controlul:

```cpp
// C++
#include <memory>

std::unique_ptr<ClasaMea> obiect = std::make_unique<ClasaMea>();
// obiectul este distrus automat la iesirea din scope
```

În Python, gestionarea memoriei poate fi optimizată prin utilizarea atributului `__slots__`, care elimină dicționarul intern al obiectului, reducând astfel consumul de memorie:

```cpp
class ObiectCompact:
```

```
2     __slots__ = ['x', 'y']
```

Această tehnică este utilă mai ales în aplicații care creează un număr foarte mare de obiecte.

10.4 Tehnici de depanare

Oricât de experimentat ar fi un programator, erorile fac parte din viața oricărui proiect software. Învățarea unor tehnici eficiente de depanare este esențială nu doar pentru a rezolva problemele mai repede, ci și pentru a înțelege mai profund comportamentul aplicațiilor. Indiferent că folosim C++ sau Python, depanarea presupune o combinație între logică, cunoștințe tehnice și, uneori, o doză sănătoasă de răbdare.

10.4.1 Mesaje de jurnalizare și afișări intermediare

Una dintre cele mai simple, dar eficiente tehnici de depanare, este inserarea de mesaje în cod care să ne arate ce se întâmplă în anumite puncte cheie ale execuției. În C++, acest lucru se face frecvent cu ajutorul fluxului `std::cout`:

```
1 // C++
2 void proceseaza_datele(int numar) {
3     std::cout << "Am primit valoarea: " << numar << std::endl;
4     // ... alte operatii
5 }
```

În Python, echivalentul este funcția `print()`:

```
1 # Python
2 def proceseaza_datele(numar):
3     print(f"Am primit valoarea: {numar}")
4     # ... alte operatii
```

Deși aceste mesaje ajută în etapa inițială de depanare, ele trebuie folosite cu măsură. Este recomandat ca într-un cod de producție să se utilizeze un sistem de jurnalizare mai flexibil (cum ar fi `logging` în Python sau `logging` cu nivele în C++), pentru a controla mai bine nivelul și destinația mesajelor.

10.4.2 Utilizarea debugger-elor

Debugger-ele permit rularea programelor pas cu pas, inspectarea variabilelor, monitorizarea stivei de apeluri și oprirea execuției la puncte de interes definite de programator.

În C++, cele mai populare debugger-e sunt:

- **GDB** – folosit în medii Unix/Linux.
- **LLDB** – alternativa modernă, folosită mai ales pe macOS.
- **Visual Studio Debugger** – integrat în mediul Visual Studio, foarte avansat pentru C++.

Pentru a utiliza GDB, programul trebuie compilat cu informații de depanare:

```
g++ -g program.cpp -o program
gdb ./program
```

O sesiune simplă în GDB ar putea arăta astfel:

```
(gdb) break functia_critica
(gdb) run
(gdb) print variabila
(gdb) next
(gdb) continue
```

În Python, depanarea este integrată în limbaj. Modulul `pdb` permite o experiență similară:

```
import pdb

def calculeaza_suma(a, b):
    pdb.set_trace()
    return a + b

rezultat = calculeaza_suma(3, 5)
```

La rulare, execuția se va opri în momentul în care se apelează `set_trace()`, permițând inspectarea variabilelor și navigarea prin cod cu comenzi precum `n` (next), `s` (step), `p` (print), `c` (continue).

10.4.3 Verificarea invariatelor și aserțiuni

Pentru a prinde erori cât mai devreme, o practică foarte utilă este verificarea explicită a anumitor condiții logice despre care știm că trebuie să fie adevărate – adică invariante. În C++, putem folosi macro-ul `assert` din biblioteca standard:

```
#include <cassert>

void seteaza_varsta(int varsta) {
    assert(varsta >= 0 && varsta < 130);
    // ...
}
```

În Python, avem o construcție similară:

```
def seteaza_varsta(varsta):
    assert 0 <= varsta < 130
    # ...
```

Aserțiunile nu trebuie văzute ca o formă de tratare a erorilor de intrare, ci ca o metodă de a documenta și verifica presupunerile interne ale codului.

10.4.4 Analiza comportamentului în timp de execuție

În anumite cazuri, depanarea nu se referă doar la erori de logică, ci la probleme legate de performanță, consum de memorie sau blocaje. Pentru astfel de situații, avem la dispoziție unelte specializate.

În C++:

- **Valgrind** – excelent pentru detectarea scurgerilor de memorie și accesărilor nevalide.
- **gprof**, **perf** – pentru profilarea performanței.

Exemplu de rulare cu Valgrind:

```
valgrind --leak-check=full ./program
```

În Python:

- Modulul `cProfile` este util pentru a vedea unde petrece programul cel mai mult timp.
- Biblioteci precum `memory_profiler` pot arăta exact câtă memorie este consumată de fiecare funcție.

Exemplu de profilare:

```
import cProfile

def program_principal():
    # ...

cProfile.run('program_principal()')
```

10.4.5 Testare ca suport pentru depanare

O tehnică modernă care ajută semnificativ în depanare este scrierea de teste automate. Testele nu doar că verifică dacă o parte a programului funcționează corect, ci servesc ca punct de referință atunci când ceva se strică.

În C++, putem folosi biblioteci precum `GoogleTest`, iar în Python modulul `unittest` sau `pytest` sunt opțiuni populare.

Un test simplu în Python:

```
import unittest

class TestSuma(unittest.TestCase):
    def test_suma_corecta(self):
        self.assertEqual(3 + 5, 8)

unittest.main()
```

11. Concluzii

Programarea orientată pe obiecte (OOP) este o paradigmă fundamentală în dezvoltarea software modern, iar Python și C++ sunt două dintre cele mai proeminente și utilizate limbaje de programare care adoptă această paradigmă. Fiecare dintre aceste limbaje vine cu propriile particularități, avantaje și limitări, fiind utilizate într-o gamă largă de aplicații, în funcție de cerințele specifice ale proiectelor.

Python este un limbaj interpretat, recunoscut pentru ușurința de învățare și utilizare. Sintaxa sa simplă, bazată pe indentare și utilizarea cuvintelor cheie clare, îl face accesibil începătorilor și eficient pentru dezvoltarea rapidă. În schimb, C++ este un limbaj compilat, derivat din limbajul C, oferind performanțe superioare și control direct asupra resurselor hardware, ceea ce îl face potrivit pentru aplicații complexe și performante.

Sintaxa Python este intuitivă și minimalistă, facilitând înțelegerea rapidă și reducând erorile cauzate de codul prea complex sau detaliat. Acest lucru contribuie semnificativ la adoptarea sa rapidă în mediul academic și în proiecte de prototipare rapidă. În contrast, sintaxa limbajului C++ este mai detaliată și complexă, necesitând declararea explicită a tipurilor de date, fapt care, deși crește complexitatea, oferă mai mult control și precizie.

În ceea ce privește gestionarea memoriei, Python folosește colectarea automată a deșeurilor (garbage collection), facilitând dezvoltarea aplicațiilor fără a necesita atenție explicită asupra gestionării memoriei. Acest avantaj, însă, vine cu un cost al performanței, în special în aplicațiile cu cerințe ridicate de resurse. În schimb, C++ permite controlul manual al memoriei, oferind dezvoltatorilor posibilitatea de a optimiza utilizarea resurselor hardware prin intermediul pointerilor și al pointerilor inteligenți (smart pointers), dar implică și un nivel crescut de responsabilitate și complexitate în scrierea codului.

C++ oferă o performanță superioară datorită procesului de compilare, generând cod executabil eficient și rapid. Această caracteristică face din C++ alegerea ideală pentru dezvoltarea aplicațiilor critice din punct de vedere al vitezei, precum jocuri video, sisteme de operare, aplicații embedded și simulări complexe. Python, în schimb, sacrifică performanța în favoarea flexibilității și a rapidității dezvoltării, ceea ce îl face ideal pentru aplicații web, scripting, analiză de date, machine learning și prototipare rapidă.

Diferențele în tipizarea datelor reprezintă un alt aspect esențial. Python utilizează o tipizare dinamică, permițând modificarea tipului unei variabile în timpul execuției, ceea ce oferă flexibilitate și viteză în dezvoltarea prototipurilor. C++, pe de altă parte, utilizează tipizare statică, oferind mai multă stabilitate și siguranță în timpul execuției, prevenind erorile legate de incompatibilități între tipuri, dar necesitând planificare atentă și rigurozitate din partea programatorului.

Python beneficiază de o comunitate largă și activă, care contribuie continuu la dezvoltarea unui ecosistem extins de biblioteci și framework-uri pentru o varietate impresionantă de aplicații. Această comunitate vastă facilitează dezvoltarea și menținerea proiectelor Python și oferă suport excelent pentru începători și experți deopotrivă. În schimb, C++ se bazează pe o comunitate solidă de dezvoltatori experimentați, orientați spre performanță și aplicații critice, care promovează practici riguroase și o abordare profundă asupra optimizării software.

Alegerea între Python și C++ depinde fundamental de obiectivele și cerințele proiectului. Python este preferat atunci când prioritatea o reprezintă dezvoltarea rapidă, prototiparea și flexibilitatea în implementare, în special în domenii precum dezvoltarea web, data science și automatizarea sarcinilor repetitive. În contrast, C++ este opțiunea evidentă pentru proiectele în care performanța, eficiența și controlul hardware sunt esențiale.

Astfel, stăpânirea ambelor limbaje și cunoașterea aprofundată a particularităților lor contribuie la formarea unui dezvoltator software complet și adaptabil, capabil să abordeze o gamă variată de provocări în dezvoltarea de software modern și performant.

Bibliografie

[1] L. Ammeraal, *STL (Standard Template Library) for C++ Programmers*. John Wiley & Sons Inc, 1996.

[2] B. Eckel, *Thinking in C++*, 2nd Ed. Prentice Hall, 2003, http://www.odioworks.com/46-Bruce_Eckel's_Free_Electronic_Books.html.

[3] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[4] M. T. Goodrich, *Data Structures and Algorithms in C++*, 2nd Ed. Wiley, 2011.

[5] U. Kirch-Priz and P. Prinz, *A Complete Guide to Programming in C++*. Jones and Bartlett, http://www.lmpt.univ-tours.fr/~volkov/C++.pdf.

[6] D. Logofătu, *Algoritmi fundamentali în C++. Aplicații*, 1st Ed. Iași: Editura Polirom, 2007.

[7] P. Müller, "Introduction to object-oriented programming using c++," 1997, globewide Network Academy (GNA), http://www.desy.de/gna/html/cc/Tutorial/tutorial.html.

[8] D. Nesteruk, *Design Patterns in Modern C++20: Reusable Approaches for Object-Oriented Software Design*, 2nd Ed. Apress, 2021.

[9] M. Popa and M. Popa, *Elemente de algoritmi și limbaje de programare*. Editura Universității din București, 2005.

[10] D. Roman, *Ingineria programării obiectuale*. Cluj Napoca: Editura Albastră, 2008.

[11] H. Schildt, *C - Manual complet*. București: Editura Teora, 2000.

[12] H. Schildt, *C++ - Manual complet*. București: Editura Teora, 2001.

[13] B. Stroustrup, *The C++ programming language*, 3rd Ed. Addison-Wesley, 1997.

[14] B. Stroustrup, *The C++ Programming Language*, 4th Ed. Pearson Education, 2013.

[15] D. Ungureanu, *Programare obiectuală folosind C++*. Brașov: Editura Transilvania, 2009.

[16] D. Ungureanu, *C/C++ Programming. Learn by Examples.* Brașov: Editura Universității Transilvania, 2012.

[17] G. van Rossum, "Python programming language," *Proceedings of the 2007 USENIX Annual Technical Conference, USENIX ATC 2007, Santa Clara, CA, USA, June 17-22, 2007*, J. Chase and S. Seshan, Eds. USENIX, 2007.

[18] M. D. Yaharia, *Structuri de date și algoritmi - exemple în limbajele C și C++.* Cluj-Napoca: Editura Microinformatica, 2002.