UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Concepte OOP în C++

Dan GOTA Alexandra FANCA
Adela POP Honoriu VALEAN

UTPRESS
Cluj-Napoca, 2022
ISBN 978-606-737-590-9

Dan Ioan GOTA
Alexandra FANCA
Adela POP
Honoriu VĂLEAN

CONCEPTE OOP ÎN C++

UTPRESS
Cluj - Napoca, 2022
ISBN 978-606-737-590-9

Editura U.T.PRESS
Str. Observatorului nr. 34
400775 Cluj-Napoca
Tel.: 0264-401.999
e-mail: utpress@biblio.utcluj.ro
http://biblioteca.utcluj.ro/editura

Director: ing. Dan Colțea

Recenzia: Prof.dr.ing. Liviu Miclea
Conf.dr.ing. Ovidiu Stan

Pregătire online: Gabriela Groza

ISBN 978-606-737-590-9
Bun de tipar: 22.09.2022

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Cuprins

Capitolul 0: Scurtă introducere C++

C++ este un limbaj de programare de nivel mediu dezvoltat de Bjarne Stroustrup începând cu 1979 la Bell Labs. C++ rulează pe o varietate de platforme, cum ar fi Windows, Mac OS și diferite versiuni de UNIX. Această carte C++ adoptă o abordare simplă și practică pentru a descrie conceptele C++ pentru începători până la inginerii de software avansați.

C++ este un MUST pentru studenții și profesioniștii care lucrează pentru a deveni ingineri software. In cele ce urmează sunt enumerate câteva dintre avantajele cheie ale învățării limbajului C++:

- Limbajul C++ este strâns legat de echipamentele hardware, astfel este posibilă programarea la un nivel scăzut, ceea ce oferă mult control în ceea ce privește gestionarea memoriei, performanță mai bună și, în sfârșit, o dezvoltare software robustă.
- Programarea C++ oferă o înțelegere clară despre programarea orientată pe obiecte.
- C++ este cel mai utilizat limbaj în programarea de aplicații și de sisteme [1].

I. Utilitatea limbajul C++

După cum am menționat anterior, C++ este unul dintre cele mai utilizate limbaje de programare. Are prezența sa în aproape toate domeniile dezvoltării software. În continuare sunt enumerate câteva dintre ele:

- ***Dezvoltare de software pentru aplicații*** - Programarea C++ a fost folosită în dezvoltarea aproape tuturor sistemelor de operare majore precum Windows, Mac OSX și Linux. În afară de sistemele de operare, partea centrală a multor browsere precum Mozilla, Firefox și Chrome au fost scrise folosind C++. Acesta a fost, de asemenea, folosit și în dezvoltarea celui mai popular sistem de gestionare a bazelor de date numit MySQL.
- ***Dezvoltarea limbajelor de programare*** - C++ a fost utilizat pe scară largă în dezvoltarea de noi limbaje de programare precum C#, Java, JavaScript, Perl, C Shell UNIX, PHP și Python și Verilog etc.
- ***Programare de calcul*** - C++ este cel mai bun prieten al oamenilor de știință datorită vitezei rapide și a eficienței de calcul.
- ***Dezvoltarea jocurilor*** - C++ este extrem de rapid, ceea ce le permite programatorilor să efectueze programare procedurală pentru funcții intensive de CPU și oferă un control mai mare asupra hardware-ului, astfel a fost utilizat pe scară largă în dezvoltarea motoarelor de jocuri.
- ***Sisteme încorporate(embedded)*** - C++ este utilizat intens în dezvoltarea de aplicații medicale și de inginerie, cum ar fi software-uri pentru aparate RMN, sisteme CAD/CAM de ultimă generație etc.

Limbajul C++ a fost creat pentru a aduce orientarea obiectelor în limbajul de programare C, care este deja unul dintre cele mai sofisticate limbaje de programare disponibile. Punctul central al programării pur orientate pe obiecte este de a scrie cod care creează un obiect cu anumite caracteristici și metode. Când creăm module C++, ne străduim să vizualizăm întregul univers ca obiecte. Un automobil, de exemplu, este un obiect cu caracteristici

particulare, cum ar fi culoarea, numărul de uși, capacitate motor și așa mai departe. Include, de asemenea, funcții precum accelerarea, frânarea și așa mai departe [2].

II. Noțiuni de bază ale limbajului C++

Programarea orientată pe obiecte se bazează pe câteva noțiuni de bază. In cele ce urmează sunt prezentate pe scurt aceste noțiuni de bază.

Obiect

Acesta este elementul fundamental al programării orientate pe obiecte. Adică, atât datele, cât și funcțiile de operare a datelor sunt grupate împreună ca obiect/obiecte.

Clasă

Un plan/schema pentru un obiect este definit atunci când este definită o clasă. Aceasta nu descrie date, dar definește ce înseamnă numele clasei, adică ce este un obiect de clasă și ce operațiuni pot fi efectuate asupra acestuia.

Abstractizare

Abstractizarea datelor este definită ca furnizarea de informații importante către lumea exterioară în timp ce se maschează detaliile de fundal, adică reprezentând informațiile necesare într-un program fără a afișa specificul.

Un sistem de baze de date, de exemplu, ascunde anumite aspecte ale stocării, creării și întreținerii datelor. În mod similar, clasele C++ expun multe metode lumii exterioare fără a dezvălui informații interne despre acele funcții sau date.

Încapsularea

Încapsularea este procesul de punere a datelor și a funcțiilor care operează cu acestea în aceeași locație. Nu este întotdeauna evident ce funcții funcționează pe ce variabile atunci când aveți de-a face cu limbaje procedurale, dar programarea orientată pe obiecte vă oferă un cadru pentru a pune împreună datele și funcțiile relevante în același obiect.

Moștenirea

Reutilizarea codului este una dintre cele mai benefice proprietăți ale programării orientate pe obiecte. După cum sugerează și numele, moștenirea este procesul de creare a unei noi clase dintr-o clasă existentă cunoscută sub numele de clasă de bază. Noua clasă este cunoscută sub numele de clasă derivată. Aceasta este o noțiune crucială în programarea orientată pe obiect, deoarece ajută la reducerea dimensiunii codului.

Polimorfismul

Polimorfismul este capacitatea de a folosi un operator sau o funcție în mai multe moduri sau de a oferi diferite semnificații sau funcții operatorilor sau funcțiilor. Termenul „poli” înseamnă „mulți”. Polimorfismul se referă la o singură funcție sau operator care poate fi utilizat într-o varietate de moduri, în funcție de context.

Supraîncărcare

Polimorfismul include și ideea de supraîncărcare cunoscut sub denumirea de Overloading. Este considerat a fi supraîncărcat atunci când un operator sau o funcție existentă este pus să lucreze pe un nou tip de date [3].

Capitolul 1: Clase și obiecte în C++

O clasă, specifică forma unui obiect combinând reprezentarea datelor și metodele de modificare a acestor date într-un singur pachet. Membrii unei clase sunt datele și metodele care alcătuiesc clasa.

I. Definirea claselor în C++

Când se creează o clasă, în esență se creează un plan pentru un tip de date. Aceasta nu descrie date, dar definește ce înseamnă numele clasei, adică din ce va fi alcătuit un obiect de clasă și ce acțiuni pot fi efectuate asupra acestuia.

O definiție de clasă începe cu cuvântul cheie ***class***, apoi numele clasei și, în final, corpul clasei, care este conținut de o pereche de paranteze. Un punct și virgulă sau o serie de declarații trebuie să vină după o definiție de clasă.

De exemplu, s-a definit tipul de date Cutie folosind clasa de cuvinte cheie după cum urmează:

```
class Cutie {
public:
double lungime;   // Lungimea cutiei
double lățime;  // Lățimea cutiei
double înălțime;   // Înălțimea cutiei
};
```

Caracteristicile de acces ale membrilor clasei care o urmează sunt determinate de cuvântul cheie ***public***. Oriunde în sfera obiectului de clasă, un membru public poate fi accesat din afara clasei. De

asemenea, membrii unei clase pot fi setați secreți sau protejați, lucru detaliat într-o secțiune ulterioară.

II. Definirea obiectelor în C++

O clasă servește ca model pentru lucruri, prin urmare, un obiect este în esență format dintr-unul. Obiectele unei clase sunt declarate în același mod în care sunt declarate variabilele de tipuri fundamentale. Următoarele instrucțiuni declară două obiecte de tip Cutie.

```
Cutie Cutie1;           // Declarare Cutie1 de
tipul Cutie
Cutie Cutie2;           // Declarare Cutie2 de
tipul Cutie
```

Ambele obiecte Cutie1 și Cutie2 vor avea propria lor copie a membrilor datelor.

III. Accesarea membrilor de date

Operatorul de acces direct, „.”, al membrilor poate fi utilizat pentru a accesa membrii datelor publice ai obiectelor unei clase. Să încercăm următorul exemplu pentru a clarifica lucrurile. *pentru accesarea codului consultați Anexa1.

```cpp
#include <iostream>
using namespace std;

class Cutie {
public:
    double lungime;   // Lungimea cutiei
    double latime;  // Latimea cutiei
    double inaltime;   // Inaltimea cutiei
};

int main() {
    Cutie Cutie1;         // Declararea Cutie 1 de tip Cutie
    Cutie Cutie2;         // Declararea Cutie 2 de tip Cutie
    double volumCutie = 0.0;      // Stocare volum cutie in variabila volumCutie

    Cutie1.inaltime = 5.0;
    Cutie1.lungime = 6.0;
    Cutie1.latime = 7.0;

    Cutie2.inaltime = 10.0;
    Cutie2.lungime = 12.0;
    Cutie2.latime = 13.0;

    // Calcul volum pentru cutie1
    volumCutie = Cutie1.inaltime * Cutie1.lungime * Cutie1.latime;
    std::cout << "Volum pentru Cutie1 : " << volumCutie << std::endl;

    // Calcul volum pentru cutie2
    volumCutie = Cutie2.inaltime * Cutie2.lungime * Cutie2.latime;
    std::cout << "Volum pentru Cutie2 : " << volumCutie << std::endl;
    return 0;
}
```

Când codul precedent este compilat și rulat, se obține următorul rezultat:

```
Volum pentru Cutie1: 210
Volum pentru Cutie2: 1560
```

Este vital să știți că operatorul de acces direct la membri nu poate accesa membrii privați sau protejați. Vom descoperi cum să obținem acces la utilizatorii secreți și protejați [4].

IV. Clase și Obiecte în detaliu

Până acum, s-au prezentat elementele fundamentale ale claselor și obiectelor in C++. Există câteva subiecte mai fascinante legate de clasele și obiectele C++ care sunt analizate în secțiunile următoare.

IV.I Funcții de membru de clasă

O funcție de membru de clasă este o funcție care, ca orice altă variabilă, are definiția sau prototipul în declarația de clasă. Are acces la toți membrii clasei pentru care este membru și poate acționa asupra oricărui obiect al clasei respective.
Să folosim o funcție de membru pentru a accesa membrii unei clase create anterior, mai degrabă decât să le accesăm direct.

```
class Cutie {
public:
    double lungime;    // Lungimea cutiei
    double latime;   // Latimea cutiei
    double inaltime;    // Inaltimea cutiei
    double getVolum(void); //Returneaza volumul cutiei
};
```

Operatorul de rezoluție a domeniului de aplicare : poate fi utilizat pentru a specifica funcții membre fie în cadrul definiției clasei, fie separat.
Chiar dacă nu utilizați specificatorul inline, definirea unei funcții membru în definiția clasei definește funcția inline. Alternativ, puteți utiliza metoda Volum() așa cum se vede mai jos.

```
class Cutie {
public:
    double lungime;    // Lungimea cutiei
    double latime;   // Latimea cutiei
    double inaltime;    // Inaltimea cutiei
    double getVolum(void){
        return lungime*latime*inaltime;
    }
};
```

Puteți utiliza operatorul de rezoluție a domeniului (::) pentru a declara aceeași funcție în afara clasei, dacă doriți.

```
double Cutie::getVolum(void) {
   return lungime * latime * inaltime;
}
```

Singurul lucru de reținut este că trebuie să utilizați numele clasei înaintea operatorului (::). O funcție membru va fi apelată pe un obiect cu operatorul punct (.) și va manipula numai datele referitoare la acel obiect după cum urmează.

```
Cutie cutie1;           // Crearea obiectului
cutie1

cutie1.getVolum();//Apelarea functiei membru
pentru obiectul cutie1
```

Să folosim tehnicile de mai sus pentru a stabili și a obține valoarea diferiților membri ai clasei. *pentru accesarea codului consultați Anexa2.

```cpp
#include <iostream>
using namespace std;
class Cutie {
   public:
      double lungime;           // lungimea cutiei
      double latime;          // latimea cutiei
      double inaltime;          // inaltimea cutiei
      // Prototipe functii membru
      double getVolum(void);
      void setLungime( double len );
      void setLatime( double bre );
      void setInaltime( double hei );
};
// Member functions definitions
double Cutie::getVolum(void) {
   return lungime * inaltime * latime;
}
void Cutie::setLungime( double len ) {
   lungime = len;
}
void Cutie::setLatime( double bre ) {
   latime = bre;
}
void Cutie::setInaltime( double hei ) {
   inaltime = hei;
}
```

```cpp
// Functia Main
int main() {
   Cutie Cutie1;                // Declarare Cutie1 de tip Cutie
   Cutie Cutie2;                // Declarare Cutie2 de tip Cutie
   double volum = 0.0;      // Declarare cu initializare variabila volum
   // specificatii cutie1
   Cutie1.setLungime(6.0);
   Cutie1.setLatime(7.0);
   Cutie1.setInaltime(5.0);
   // specificatii cutie2
   Cutie2.setLungime(12.0);
   Cutie2.setLatime(13.0);
   Cutie2.setInaltime(10.0);
   // calculare volum pt cutie1
   volum = Cutie1.getVolum();
   cout << "Volumul Cutie1 : " << volum <<endl;
   // calculare volum pt cutie2
   volum = Cutie2.getVolum();
   cout << "Volum Cutie2 : " << volum <<endl;
   return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat:

```
Volum pentru Cutie1 : 210
Volum pentru Cutie2 : 1560
```

IV.II Modificatori de acces la clasă

Un membru al clasei poate fi făcut public, privat sau protejat. In mod implicit membrii sunt declarați privați. Una dintre caracteristicile cheie ale programării orientate pe obiecte este ascunderea datelor, care permite metodelor programului să evite accesul direct la reprezentarea internă a unui tip de clasă. Porțiunile publice, private și protejate desemnate din interiorul corpului clasei definesc restricțiile de acces pentru membrii clasei. Specificatorii de acces sunt următorii termeni: ***public, private și protected***. O clasă poate avea numeroase părți etichetate ca publice, protejate sau private. Fiecare secțiune continuă în vigoare până când este observată o altă etichetă de secțiune sau acoladă de închidere a corpului clasei. Membrii și clasele au acces privat în mod implicit.

```
class Baza {
   public:
      // membrii public sunt declarati aici
   protected:
      // membrii protected sunt declarati aici
   private:
   // membrii private sunt declarati aici

};
```

IV.II.I Membrii publici

Un membru public poate fi accesat de oriunde în cadrul unui program, nu numai din cadrul clasei. După cum se vede în exemplul următor, puteți seta și primi valoarea variabilelor publice fără a utiliza funcții membre. *pentru accesarea codului consultați Anexa3.

```cpp
#include <iostream>

using namespace std;

class Linie {
   public:
      double lungime;
      void setLungime( double len );
      double getLungime( void );
};

// definirea functiilor membru
double Linie::getLungime(void) {
   return lungime ;
}

void Linie::setLungime( double len) {
   lungime = len;
}

// functia Main a programului
int main() {
   Linie linie;

   // setare lungime linie
   linie.setLungime(6.0);
   cout << "Lungimea liniei este: " << linie.getLungime() <<endl;

   // setare lungime fara a utiliza functii membru
   linie.lungime = 10.0; // OK: deoarece lungime este declarata public
   cout << "Lungimea liniei este: " << linie.lungime <<endl;

   return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat:

```
Lungimea liniei este: 6

Lungimea liniei este: 10
```

IV.II.II Membrii privați

Din afara clasei, o variabilă sau o funcție de membru privat nu poate fi accesată sau nici măcar vizualizată. Membrii privați pot fi accesați numai folosind funcțiile de clasă și funcții prietene (friend).
În mod implicit, toți membrii unei clase sunt privați; de exemplu, lățime este un membru privat din clasa de mai jos, ceea ce implică

faptul că, dacă nu etichetați un membru, se va presupune că este un membru privat.

```
class Cutie {
   double latime;

   public:
      double lungime;
      void setLatime( double wid );
      double getLatime( void );
};
```

După cum se vede în programul următor, declarăm datele în partea privată și funcțiile asociate în zona publică, astfel încât acestea să poată fi invocate din afara clasei.

```
1   #include <iostream>
2
3   using namespace std;
4
5   class Cutie {
6   public:
7       double lungime;
8       void setLatime(double wid);
9       double getLatime(void);
10
11  private:
12      double latime;
13  };
14
15  // definirea functiilor membru
16  double Cutie::getLatime(void) {
17      return latime;
18  }
19
20  void Cutie::setLatime(double wid) {
21      latime = wid;
22  }
```

```
// functia Main din program
int main() {
    Cutie cutie;

    // setare lungime cutie fara functii membru
    cutie.lungime = 10.0; // OK: deoarece lungimea este declarata public
    cout << "Lungimea cutiei este: " << cutie.lungime << endl;

    // setare latime cutie fara functii membru
    // cutie.latime = 10.0; // Error: latimea este declarata private
    cutie.setLatime(10.0);  // utilizarea functiilor membru pentru setarea latimii.
    cout << "Latimea cutiei este: " << cutie.getLatime() << endl;

    return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat:

```
Lungimea cutiei este: 10

Latimea cutiei este: 10
```

IV.II.III Membrii protejați (protected)

O variabilă sau o funcție de membru protejat este comparabilă cu un membru privat, dar are avantajul suplimentar de a fi accesibilă prin clase derivate, care sunt clase copil.

În capitolul următor, veți afla despre clasele derivate și despre moștenire. Deocamdată, puteți privi următorul exemplu, în care am creat o clasă copil CutieMica dintr-o clasă părinte Cutie.

Următorul exemplu este identic cu cel anterior, cu excepția faptului că orice funcție membru din clasa sa derivată CutieMica va avea acces la membrul lățime.

```cpp
#include <iostream>
using namespace std;

class Cutie {
protected:
    double latime;
};

class CutieMica :Cutie { // Cutia mica este derivata din clasa Cutie.
public:
    void setLatimeMica(double wid);
    double getLatimeMica(void);
};

// Functiile membru ale clasei copil
double CutieMica::getLatimeMica(void) {
    return latime;
}

void CutieMica::setLatimeMica(double wid) {
    latime = wid;
}

// Functia Main a programului
int main() {
    CutieMica cutie;

    // setare latime cutie fara functii membru
    cutie.setLatimeMica(5.0);
    cout << "Latime cutie este: " << cutie.getLatimeMica() << endl;

    return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat:

```
Latime cutie este: 5
```

IV.III Constructori și destructori

Când este creat un nou obiect al unei clase, este invocat un constructor de clasă. Un destructor este o anumită funcție care este rulată atunci când un obiect este creat și apoi eliminat.

Un constructor va avea același nume ca și clasa și nu va returna nicio valoare, nici măcar void. Constructorii sunt folositori atunci când vine vorba de stabilirea valorilor inițiale pentru variabilele membre. Noțiunea de constructor este explicată în exemplul următor.

```
#include <iostream>

using namespace std;

class Linie {
public:
    void setLungime(double len);
    double getLungime(void);
    Linie();  // aceasta este un constructor
private:
    double lungime;
};

// functiile membru inclusiv constructorul
Linie::Linie(void) {
    cout << "Obiectul este creat" << endl;
}
void Linie::setLungime(double len) {
    lungime = len;
}
double Linie::getLungime(void) {
    return lungime;
}

int main() {
    Linie linie;

    // setare lungime linie
    linie.setLungime(6.0);
    cout << "Lungime linie este: " << linie.getLungime() << endl;

    return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat:

```
Obiectul este creat

Lungime linie este: 6
```

IV.III.I Constructori parametrizați

Un constructor implicit nu are niciun parametru, dar îi puteți adăuga dacă aveți nevoie. După cum se vede în exemplul următor, acest lucru vă permite să setați o valoare inițială unui obiect atunci când este creat.

```cpp
#include <iostream>

using namespace std;
class Linie {
public:
    void setLungime(double len);
    double getLungime(void);
    Linie(double len);  // aceasta este constructorul

private:
    double lungime;
};

// functiile membru
Linie::Linie(double len) {
    cout << "Obiectul este creat, lungimea = " << len << endl;
    lungime = len;
}
void Linie::setLungime(double len) {
    lungime = len;
}
double Linie::getLungime(void) {
    return lungime;
}

int main() {
    Linie linie(10.0);

    cout << "Lungime linie : " << linie.getLungime() << endl;

    linie.setLungime(6.0);
    cout << "Lungime linie : " << linie.getLungime() << endl;

    return 0;
}
```

Când codul de mai sus este compilat și executat, produce următorul rezultat:

```
Obiectul este creat, lungime = 10
Lungime linie : 10
Lungime linie : 6
```

IV.III.II Utilizarea listelor de inițializare pentru a inițializa câmpuri(fields)

Pentru a inițializa câmpurile într-un constructor parametrizat, utilizați următoarea sintaxă:

```
Linie::Linie( double len): lungime(len) {
   cout << "Obiectul este creat, lungime = " <<
len << endl;
}
```

Sintaxa de mai sus este egală cu următoarea sintaxă:

```
Linie::Linie( double len) {
   cout << "Obiectul este creat, lungime = " <<
len << endl;
   lungime = len;
}
```

Dacă aveți multe câmpuri de inițializat pentru o clasă C, cum ar fi X, Y, Z și așa mai departe, puteți utiliza aceeași sintaxă și puteți separa câmpurile cu o virgulă, după cum urmează:

```
C::C( double a, double b, double c): X(a), Y(b),
Z(c) {
   ....
}
```

IV.III.III Destructorul unei clase

Un destructor este o funcție membru specifică a unei clase care este apelată ori de câte ori un obiect din acea clasă iese din domeniul de aplicare sau atunci când expresia de ștergere este utilizată pe o referință la obiectul acelei clase.
Un destructor va avea același nume ca și clasa, dar va fi prefixat cu un simbol tilda (~) și nu va putea returna nicio valoare sau nu va putea lua niciun parametru. Destructorul este util pentru eliberarea

resurselor înainte de a părăsi un program, cum ar fi închiderea fișierelor și eliberarea memoriei.
Următorul exemplu demonstrează cum să utilizați un destructor.

```
#include <iostream>

using namespace std;
class Linie {
public:
    void setLungime(double len);
    double getLungime(void);
    Linie();    // Declarare constructor
    ~Linie();   // Declarare destructor

private:
    double lungime;
};

// Functiile membru
Linie::Linie(void) {
    cout << "Obiectul este creat" << endl;
}
Linie::~Linie(void) {
    cout << "Obiectul este sters" << endl;
}
void Linie::setLungime(double len) {
    lungime = len;
}
double Linie::getLungime(void) {
    return lungime;
}

int main() {
    Linie linie;

    // setarea lungime linie
    linie.setLungime(6.0);
    cout << "Lungime linie este: " << linie.getLungime() << endl;

    return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat:

```
Obiectul este creat

Lungime linie este: 6

Obiectul este sters
```

IV.III.IV Constructorul de copiere

Constructorul de copiere este un constructor care produce un obiect prin inițializarea acestuia folosind un obiect construit anterior din aceeași clasă.

- Inițializați un obiect dintr-un altul de același tip folosind constructorul de copiere;
- Pentru a furniza un obiect ca parametru unei funcții, copiați-l;
- Returnează un obiect dintr-o funcție prin copierea acestuia.

Dacă o clasă nu are un constructor de copiere, compilatorul creează unul.
Un constructor de copiere este necesar dacă clasa conține variabile pointer și anumite alocări dinamice de memorie. Acesta este cel mai popular tip de constructor de copiere.

```
classname (const classname &obj) {
   // corpul constructorului
}
```

Aici, obj este o referință la un obiect care este folosit pentru a inițializa un alt obiect. *pentru accesare cod consultați Anexa4.

```
1   #include <iostream>
2
3   using namespace std;
4
5   class Linie {
6
7   public:
8       int getLungime(void);
9       Linie(int len);               // constructor simplu
10      Linie(const Linie &obj);  // constructor copiere
11      ~Linie();                      // destructor
12
13  private:
14      int *ptr;
15  };
16
17  // functii membru
18  Linie::Linie(int len) {
19      cout << "Constructor normal - alocare ptr" << endl;
20
21      // alocare memorie pentru pointer;
22      ptr = new int;
23      *ptr = len;
24  }
25
26  Linie::Linie(const Linie &obj) {
27      cout << "Constructor copiere - alocare ptr." << endl;
28      ptr = new int;
29      *ptr = *obj.ptr; // copiere valoare
30  }
31
32  Linie::~Linie(void) {
33      cout << "Eliberare memorie!" << endl;
34      delete ptr;
35  }
36
37  int Linie::getLungime(void) {
38      return *ptr;
39  }
40
41  void display(Linie obj) {
42      cout << "Lungime linie este: " << obj.getLungime() << endl;
43  }
44
45  int main() {
46      Linie linie(10);
47
48      display(linie);
49
50      return 0;
51  }
52
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat:

```
Constructor normal - alocare ptr
Constructor copiere - alocare ptr.
Lungime linie este: 10
Eliberare memorie!
Eliberare memorie!
```

Să ne uităm la exemplul identic cu o mică diferență: vom construi un alt obiect folosind un obiect existent de același tip.

```
1   #include <iostream>
2
3   using namespace std;
4
5   class Linie {
6   public:
7       int getLungime(void);
8       Linie(int len);              // Constructor simplu
9       Linie(const Linie &obj);  // Constructor copiere
10      ~Linie();                    // Destructor
11
12  private:
13      int *ptr;
14  };
```

```
16  // Member functions definitions including constructor
17  Linie::Linie(int len) {
18      cout << "Constructor normal - alocare ptr" << endl;
19
20      // alocam memorie pentru pointer;
21      ptr = new int;
22      *ptr = len;
23  }
24
25  Linie::Linie(const Linie &obj) {
26      cout << "Constructor copiere - alocare ptr." << endl;
27      ptr = new int;
28      *ptr = *obj.ptr; // copiaza valoarea
29  }
30
31  Linie::~Linie(void) {
32      cout << "Eliberare memorie!" << endl;
33      delete ptr;
34  }
35
36  int Linie::getLungime(void) {
37      return *ptr;
38  }
39
40  void display(Linie obj) {
41      cout << "Lungime linie este: " << obj.getLungime() << endl;
42  }
43
44  int main() {
45
46      Linie linie1(10);
47
48      Linie linie2 = linie1; // se apeleaza de asemenea constructorul de copiere
49
50      display(linie1);
51      display(linie2);
52
53      return 0;
54  }
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat:

```
Constructor normal - alocare ptr
Constructor copiere - alocare ptr.
Constructor copiere - alocare ptr.
Lungime linie este: 10
Eliberare memorie!
Constructor copiere - alocare ptr.
Lungime linie este: 10
Eliberare memorie!
Eliberare memorie!
Eliberare memorie!
```

IV.IV Funcții prieten (friend)

O funcție prieten are acces deplin la membrii privați și protejați ai clasei. Funcția prieten a unei clase este declarată în afara domeniului de aplicare al clasei, dar are acces la toate elementele private și protejate ale clasei. Prietenii nu sunt funcții membre, în ciuda faptului că prototipurile pentru funcțiile prieten apar în specificația clasei. O funcție, un șablon de funcție sau o funcție de membru poate fi un prieten, la fel ca o clasă sau un șablon de clasă, caz în care întreaga clasă și membrii săi sunt prieteni. Pentru a desemna o funcție ca prieten al unei clase, utilizați cuvântul cheie ***friend*** înainte de prototipul funcției în declarația clasei, așa cum se arată mai jos.

```
class Cutie {
   double latime;

   public:
      double lungime;
      friend void printareLatime( Cutie cutie );
      void setLatime( double wid );
};
```

Pentru a declara toate funcțiile membre ale clasei ClasaDoi ca prieteni ai clasei ClasaUnu, plasați următoarea declarație în definiția clasei ClasaUnu.

```
friend class ClasaDoi;
```

Luând în considerare următorul program. *pentru accesare cod consultați Anexa5.

```cpp
#include <iostream>

using namespace std;

class Cutie {
    double latime;

public:
    friend void printeazaLatime(Cutie box);
    void setLatime(double wid);
};

// Member function definition
void Cutie::setLatime(double wid) {
    latime = wid;
}

// Nota: printeazaLatime() nu este functie membru in nici o clasa.
void printeazaLatime(Cutie cutie) {
    /* Deoarece printeazaLatime() este un prieten al clasei Cutie, poate accesa direct
    orice membru al clasei Cutie */
    cout << "Latime cutie este: " << cutie.latime << endl;
}

int main() {
    Cutie cutie;

    // seteaza latime cutie fara functii membru
    cutie.setLatime(10.0);

    // utilizeaza functia prieten pentru afisarea latimii.
    printeazaLatime(cutie);

    return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat.

```
Latime cutie este: 10
```

IV.V Pointerul this

Fiecare obiect are un indicator(pointer) unic care se referă la obiectul real.

În C++, fiecare obiect are acces la propria sa adresă printr-un pointer esențial cunoscut sub numele de pointer this. Toate funcțiile membre iau acest pointer ca argument implicit. Ca rezultat, aceasta poate fi folosită pentru a se referi la obiectul care apelează într-o funcție membru.

Prietenii(friend) nu sunt membri ai unei clase, prin urmare nu există acest indicator în funcțiile de prieteni. Acest indicator este disponibil numai pentru funcțiile membre.

Pentru a înțelege în continuare noțiunea pointerului this, luați în considerare următorul exemplu. *pentru accesare cod consultați Anexa6.

```
#include <iostream>

using namespace std;

class Cutie {
public:
    // definire constructor
    Cutie(double l = 2.0, double b = 2.0, double h = 2.0) {
        cout << "Apelare Constructor." << endl;
        lungime = l;
        latime = b;
        inaltime = h;
    }
    double Volum() {
        return lungime * latime * inaltime;
    }
    int comparare(Cutie cutie) {
        return this->Volum() > cutie.Volum();
    }

private:
    double lungime;      // lungime cutie
    double latime;     // latime cutie
    double inaltime;      // inaltime cutie
};

int main(void) {
    Cutie Cutie1(3.3, 1.2, 1.5);     // Declarare Cutie1
    Cutie Cutie2(8.5, 6.0, 2.0);     // Declarare Cutie2

    if (Cutie1.comparare(Cutie2)) {
        cout << "Cutie2 este mai mica decat Cutie1" << endl;
    }
    else {
        cout << "Cutie2 este egala sau mai mare decat Cutie1" << endl;
    }

    return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat.

```
Apelare Constructor.
Apelare Constructor.
Cutie2 este egala sau mai mare decat Cutie1
```

IV.VI Membrii statici ai unei clase

Membrii de date și membrii funcției unei clase pot fi ambii declarați statici. Cuvântul cheie static poate fi folosit pentru a face membrii clasei statici. Când desemnăm un membru al clasei ca fiind static, indicăm faptul că indiferent de câte obiecte ale clasei sunt produse, componenta statică este duplicată o singură dată.

Toate obiectele din clasă au un membru static. Dacă nu este prezentă nicio inițializare suplimentară, toate datele statice sunt setate la zero atunci când este creat primul obiect. Nu putem pune cuvântul cheie în declarația de clasă, dar o putem inițializa în afara acesteia, așa cum se arată în exemplul de mai jos, prin redeclararea variabilei statice și folosind operatorul de rezoluție (scope):: pentru a determina cărei clase îi aparține.

Să încercăm următorul exemplu pentru a înțelege conceptul de membri de date statice. *pentru a accesa codul de mai jos accesați Anexa 7.

```cpp
#include <iostream>

using namespace std;

class Cutie {
public:
    static int contor;

    // Definire constructor
    Cutie(double l = 2.0, double b = 2.0, double h = 2.0) {
        cout << "Constructor apelat." << endl;
        lungime = l;
        latime = b;
        inaltime = h;

        // se incrementeaza contor de fiecare data cand un obiect este creat
        contor++;
    }
    double Volum() {
        return lungime * latime * inaltime;
    }

private:
    double lungime;
    double latime;
    double inaltime;
};

// Initializarea membrului static din clasa Cutie
int Cutie::contor = 0;

int main(void) {
    Cutie Cutie1(3.3, 1.2, 1.5);    // Declarare cutie1
    Cutie Cutie2(8.5, 6.0, 2.0);    // Declarare cutie2

                                    // Afisare numar total de obiecte.
    cout << "Totalul obiectelor este: " << Cutie::contor << endl;

    return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat:

```
Constructor apelat.

Constructor apelat.

Totalul obiectelor este: 2
```

IV.VII Funcții membru statice

Puteți face un membru al funcției static, definindu-l ca atare. Acest lucru îl face independent de orice obiect anume din clasă. Chiar dacă nu există instanțe ale clasei, o funcție membru statică poate fi apelată, iar funcțiile statice sunt accesibile folosind pur și simplu numele clasei și operatorul de rezoluție a domeniului ::.

Din afara clasei, o funcție membru statică poate accesa numai membri de date statice, alte funcții membre statice și orice alte funcții.

Funcțiile membre statice sunt delimitate de clasă și nu au acces la această referință a clasei. Puteți utiliza o funcție membru statică pentru a verifica dacă anumite obiecte ale clasei au fost generate sau nu.

Să încercăm următorul exemplu pentru a înțelege conceptul de membri ai funcției statice. *pentru a accesa codul consultați Anexa 8.

```cpp
#include <iostream>
using namespace std;

class Cutie {
public:
    static int contor;
    // Definirea constructor
    Cutie(double l = 2.0, double b = 2.0, double h = 2.0) {
        cout << "Constructor apelat." << endl;
        lungime = l;
        latime = b;
        inaltime = h;
        // se incrementeaza la crearea fiecarui obiect nou
        contor++;
    }
    double Volum() {
        return lungime * latime * inaltime;
    }
    static int getContor() {
        return contor;
    }
private:
    double lungime;
    double latime;
    double inaltime;
};
// Initializarea membrului static din clasa Cutie
int Cutie::contor = 0;

int main(void) {
    // Afisare valoare contor initiala de catre obiecte.
    cout << "Valoare initiala contor: " << Cutie::getContor() << endl;

    Cutie Cutie1(3.3, 1.2, 1.5);
    Cutie Cutie2(8.5, 6.0, 2.0);
                                // Afisare total numar de obiecte create.
    cout << "Valoare contor finala: " << Cutie::getContor() << endl;

    return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat.

```
Valoare initiala contor: 0
Constructor apelat.
Constructor apelat.
Valoare contor finala: 2
```

Capitolul 2: Moștenirea în C++

Ideea de moștenire este una dintre cele mai semnificative din programarea orientată pe obiecte. Moștenirea ne permite să definim o clasă în termenii unei alte clase, facilitând dezvoltarea și întreținerea aplicațiilor. Acest lucru permite, de asemenea, reutilizarea funcționalității codului și un timp de implementare rapid.

În loc să se dezvolte membri de date complet noi și metode de membri atunci când stabilește o clasă, programatorul poate specifica că noua clasă să moștenească membrii unei clase existente. Vechea clasă este cunoscută ca clasa de bază, în timp ce noua clasă este cunoscută ca și clasă derivată.

Ideea de moștenire implementează o relație de tip ***"este un/este o" (is-a)***. De exemplu, un mamifer este un animal, iar un câine este un mamifer, ergo un câine este un animal și așa mai departe.

I. Clase de baza și clase derivate

O clasă poate fi derivată din mai multe clase de bază, permițându-i să moștenească date și funcționalități din numeroase surse. Pentru a indica clasa de bază pentru o clasă derivată, folosim o listă(e) de derivare a clasei. O listă de derivare a clasei ia forma și denumește una sau mai multe clase de bază [5].

```
class derived-class: access-specifier base-class
```

Unde clasa de bază este numele unei clase specificate anterior, iar specificatorul de acces este unul dintre public, protejat sau privat. Este privat în mod implicit dacă nu este utilizat specificatorul de acces.

Luați în considerare o clasă de bază Forma și clasa sa derivată Dreptunghi, după cum urmează. *pentru accesarea codului consultați Anexa 9.

```
#include <iostream>

using namespace std;

// Clasa de baza
class Forma {
public:
    void setLatime(int w) {
        latime = w;
    }
    void setInaltime(int h) {
        inaltime = h;
    }

protected:
    int latime;
    int inaltime;
};

// Clasa derivata
class Dreptunghi : public Forma {
public:
    int getArie() {
        return (latime * inaltime);
    }
};

int main(void) {
    Dreptunghi dreptunghi;

    dreptunghi.setLatime(5);
    dreptunghi.setInaltime(7);

    // se afiseaza aria obiectului.
    cout << "Aria este: " << dreptunghi.getArie() << endl;

    return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat.

```
Aria este: 35
```

II. Controlul accesului în conceptul de moștenire

O clasă derivată are acces la toți membrii non-privați ai clasei sale de bază. Ca rezultat, orice elemente ale clasei de bază, care nu ar trebui să fie disponibile pentru funcțiile membre ale clasei derivate ar trebui să fie marcate ca ***private*** în clasa de bază.

Putem rezuma diferitele tipuri de acces în funcție de cine le poate accesa, precum este prezentat în tabelul de mai jos.

Tip acces	**Public**	**Protected**	**Private**
Aceeași clasa	DA	DA	DA
Clasa derivata	DA	DA	NU
In afara clasei	DA	NU	NU

Cu excluderile enumerate mai jos, o clasă derivată moștenește toate metodele clasei de bază.

- Constructorii clasei de bază, destructorii și constructorii de copiere;
- Operatorii clasei de bază sunt supraîncărcați;
- Funcțiile friend ale clasei de bază.

III. Tipuri de moștenire

Când o clasă este derivată dintr-o clasă de bază, clasa de bază poate fi moștenită în unul din trei moduri: moștenire publică, protejată sau privată. După cum sa menționat anterior, specificatorul de acces specifică tipul de moștenire.

Moștenirea protejată și privată sunt rareori folosite, în timp ce moștenirea publică este. Următoarele principii sunt implementate atunci când se folosesc diferite tipuri de moștenire:

- Moștenire publică − Când se derivă o clasă dintr-o clasă de bază publică, membrii publici ai clasei de bază devin membri publici ai clasei derivate și membrii protejați ai clasei de bază devin membri protejați ai clasei derivate. Membrii privați ai unei clase de bază nu sunt niciodată accesibili direct dintr-o clasă derivată, dar pot fi accesați prin apeluri către membrii publici și protejați ai clasei de bază.
- Moștenire protejată - Când derivă dintr-o clasă de bază protejată, membrii publici și protejați ai clasei de bază devin membri protejați ai clasei derivate.
- Moștenire privată - Când o clasă derivată este derivată dintr-o clasă de bază privată, membrii publici și protejați ai clasei de bază devin membri privați ai clasei derivate.

IV. Moștenirea multiplă

O clasă C++ poate moșteni membri din mai multe clase, iar sintaxa extinsă este următoarea:

```
class derived-class: acces baseA, acces baseB,
...
```

Fiecare clasă de bază va avea unul dintre cele trei niveluri de acces: public, protejat sau privat, care vor fi separate printr-o virgulă, așa cum se vede mai sus. Să aruncăm o privire la un exemplu.

```cpp
#include <iostream>
using namespace std;
// Clasa de baza 1
class Forma {
public:
    void setLatime(int w) {
        latime = w;
    }
    void setInaltime(int h) {
        inaltime = h;
    }
protected:
    int latime;
    int inaltime;
};
// Clasa de baza 2
class CostZugravit {
public:
    int getCost(int aria) {
        return aria * 70;
    }
};
// Clasa Derivata
class Dreptunghi : public Forma, public CostZugravit {
public:
    int getAria() {
        return (latime * inaltime);
    }
};
int main(void) {
    Dreptunghi dreptunghi;
    int area;

    dreptunghi.setLatime(5);
    dreptunghi.setInaltime(7);
    area = dreptunghi.getAria();

    // Afișează aria obiectului.
    cout << "Aria totala: " << dreptunghi.getAria() << endl;
    // Afișează costul total pentru zugrăvirea dreptunghiului
    cout << "Cost total pentru zugravit: RON " << dreptunghi.getCost(area) << endl;
    return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat.

```
Aria totala: 35
Cost total pentru zugravit: RON 2450
```

Capitolul 3: Supraîncărcarea (Operatori și Funcții) în C++

Supraîncărcarea funcțiilor și supraîncărcarea operatorilor sunt termeni folosiți în C++ pentru a descrie capacitatea de a specifica mai multe definiții pentru un nume de funcție sau un operator în același domeniu.

O declarație supraîncărcată este una care are același nume ca o declarație declarată anterior în același domeniu, dar ambele declarații au parametri separați și definiții clar diferite (implementare).

Când apelați o funcție sau un operator supraîncărcat, compilatorul compară tipurile de argument pe care le-ați folosit pentru a invoca funcția sau operatorul cu tipurile de parametri menționate în definiții pentru a determina ce definiție să utilizați.

Rezoluția supraîncărcării este procesul de determinare a funcției sau operatorului supraîncărcat cel mai potrivit [6].

I. Supraîncărcarea funcțiilor în C++

În același domeniu, puteți avea numeroase definiții pentru același nume de funcție. Tipurile și/sau numărul de argumente din lista de argumente trebuie să fie diferite în fiecare declarație de funcție. Nu este posibilă supraîncărcarea declarațiilor de funcție care diferă doar prin tipul de returnare.

Un exemplu despre modul în care aceeași metodă print() poate fi utilizată pentru a tipări mai multe tipuri de date este prezentat in cele ce urmează. * accesați codul consultând Anexa11.

```cpp
#include <iostream>
using namespace std;

class PrintareDate {
public:
    void printare(int i) {
        cout << "Afisare valoare int: " << i << endl;
    }
    void printare(double  f) {
        cout << "Afisare valoare float: " << f << endl;
    }
    void printare(char* c) {
        cout << "Afisare valoare character: " << c << endl;
    }
};

int main(void) {
    PrintareDate pd;

    // Apelare metoda print cu parametru de tip intreg
    pd.printare(5);

    // Apelare metoda print cu parametru de tip float
    pd.printare(100.263);

    // Apelare metoda print cu parametru de tip caracter
    pd.printare("Salut din C++");

    return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat:

```
Afisare valoare int: 5
Afisare valoare float: 100.263
Afisare valoare character: Salut din C++
```

II. Supraîncărcarea operatorilor in C++

Majoritatea operatorilor încorporați în C++ pot fi redefiniți sau supraîncărcați. Ca rezultat, un programator poate utiliza, de asemenea, operatori cu tipuri definite de utilizator.

Operatorii supraîncărcați sunt funcții cu nume unice, constând din cuvântul cheie „operator” urmat de simbolul operatorului care urmează să fie definit. Un operator supraîncărcat, ca orice altă funcție, are un tip de returnare și o listă de argumente.

```
Cutie operator+(const Cutie&);
```

definește operatorul de adăugare, care poate fi folosit pentru a combina două instanțe Cutie și produce obiectul Cutie rezultat. Majoritatea operatorilor supraîncărcați pot fi clasificați ca funcții non-membre sau funcții membre de clasă. Dacă creăm funcția de mai sus ca o funcție non-membră a unei clase, trebuie să furnizăm două argumente pentru fiecare operand, așa cum se arată mai jos:

```
Cutie operator+(const Cutie&, const Cutie&);
```

Următorul exemplu utilizează o funcție membru pentru a demonstra noțiunea de supraîncărcare a operatorului. Obiectul care va invoca acest operator poate fi obținut utilizând acest operator așa cum este menționat mai jos, iar atributele sale vor fi accesibile utilizând acest obiect. * pentru accesare cod consultați Anexa12.

```cpp
#include <iostream>
using namespace std;

class Cutie {
public:
    double getVolum(void) {
        return lungime * latime * inaltime;
    }
    void setLungime(double len) {
        lungime = len;
    }
    void setLatime(double bre) {
        latime = bre;
    }
    void setInaltime(double hei) {
        inaltime = hei;
    }
```

```cpp
    // supraincarcarea operatorului + pentru a aduna doua obiecte de tipul Cutie.
    Cutie operator+(const Cutie& b) {
        Cutie cutie;
        cutie.lungime = this->lungime + b.lungime;
        cutie.latime = this->latime + b.latime;
        cutie.inaltime = this->inaltime + b.inaltime;
        return cutie;
    }

private:
    double lungime;      // lungime cutie
    double latime;       // latime cutie
    double inaltime;     // inaltime cutie
};

int main() {
    Cutie Cutie1;               // Declarare Cutie1 de tip Cutie
    Cutie Cutie2;               // Declarare Cutie2 de tip Cutie
    Cutie Cutie3;               // Declarare Cutie3 de tip Cutie
    double volum = 0.0;     // variabila stocare volum cutie

    // Specificatii cutie1
    Cutie1.setLungime(6.0);
    Cutie1.setLatime(7.0);
    Cutie1.setInaltime(5.0);

    // Specificatii cutie2
    Cutie2.setLungime(12.0);
    Cutie2.setLatime(13.0);
    Cutie2.setInaltime(10.0);
```

```
49
50          // volum cutie1
51          volum = Cutie1.getVolum();
52          cout << "Volum al Cutie1 : " << volum << endl;
53
54          // volum cutie2
55          volum = Cutie2.getVolum();
56          cout << "Volum al Cutie 2 : " << volum << endl;
57
58          // Adunarea celor doua obiecte cutie:
59          Cutie3 = Cutie1 + Cutie2;
60
61          // volum cutie3
62          volum = Cutie3.getVolum();
63          cout << "Volum al Cutie 3 : " << volum << endl;
64
65          return 0;
66      }
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat:

```
Volum al Cutie1: 210
Volum al Cutie2: 1560
Volum al Cutie3: 5400
```

III. Operatori supraîncărcabili/nesupraîncărcați

Mai jos regăsiți lista operatorilor care pot fi supraîncărcați.

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Urmează lista operatorilor, care nu pot fi supraîncărcați.

::	.*	.	?:

III.I Exemple de supraîncărcare a operatorilor

Iată câteva cazuri de supraîncărcare a operatorilor pentru a vă ajuta să înțelegeți mai bine acest concept.

III.I.I Supraîncărcarea operatorilor unari

Operatorii unari operează pe un singur operand, iar exemplele de mai jos arată cum funcționează.

- Operatorii pentru creștere (++) și decrementare (--);
- Operatorul minus (-) este un operator unar;
- Operatorul nu (!) logic.

Operatorii unari lucrează asupra obiectului pentru care au fost chemați și, în general, apar pe partea stângă a obiectului, ca în !obj, -obj și ++obj, dar pot fi folosiți și ca postfix, ca în obj++ sau obj--.

Următorul exemplu arată cum operatorul minus (-) poate fi supraîncărcat atunci când este utilizat ca prefix sau postfix.

```
#include <iostream>
using namespace std;

class Distanta {
private:
    int centimetri;
    int metri;
public:
    // constructori
    Distanta() {
        centimetri = 0;
        metri = 0;
    }
    Distanta(int f, int i) {
        centimetri = f;
        metri = i;
    }
    // afisarea distantelor
    void displayDistance() {
        cout << "Centimetri: " << centimetri << " Metri:" << metri << endl;
    }
    // supraincarcarea operatorului minus (-)
    Distanta operator- () {
        centimetri = -centimetri;
        metri = -metri;
        return Distanta(centimetri, metri);
    }
};

int main() {
    Distanta D1(11, 10), D2(-5, 11);
    -D1;                    // se apeleaza operatorul minus
    D1.displayDistance();   // se afiseaza D1
    -D2;                    // se apeleaza operatorul minus
    D2.displayDistance();   // se afiseaza D2

    return 0;
}
```

int Distanta::centimetri

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat:

```
F: -11 I:-10

F: 5 I:-11
```

Exemplul de mai sus clarifică noțiunea și astfel puteți utiliza o abordare similară pentru a supraîncărca Operatorii Nu logici (!).

III.I.II Supraîncărcarea operatorilor binari

Operatorii binari necesită doi parametri, iar exemplele de mai jos arată cum să-i folosim. Operatorii binari, cum ar fi operatorul de adunare (+), operatorul de scădere (-) și operatorul de împărțire (/) sunt utilizați pe scară largă.
Exemplul de mai jos demonstrează cum operatorul de adăugare (+) poate fi supraîncărcat. Operatorii de scădere (-) și de împărțire (/) pot fi, de asemenea, supraîncărcați în același mod. *accesarea codului se poate face consultand Anexa 13.

```
#include <iostream>
using namespace std;

class Cutie {
    double lungime;
    double latime;
    double inaltime;
public:
    double getVolum(void) {
        return lungime * latime * inaltime;
    }
    void setLungime(double len) {
        lungime = len;
    }
    void setLatime(double bre) {
        latime = bre;
    }
    void setInaltime(double hei) {
        inaltime = hei;
    }
    // supraincarcarea operator + pentru a aduna doua obiecte de tip Cutie
    Cutie operator+(const Cutie& b) {
        Cutie cutie;
        cutie.lungime = this->lungime + b.lungime;
        cutie.latime = this->latime + b.latime;
        cutie.inaltime = this->inaltime + b.inaltime;
        return cutie;
    }
};
```

```
int main() {
    Cutie Cutie1;                // Declarare Cutie1 de tip Cutie
    Cutie Cutie2;                // Declarare Cutie2 de tipu Cutie
    Cutie Cutie3;                // Declarare Cutie3 de tipu Cutie
    double volum = 0.0;          // Stocare date despre volum in aceasta variabila

     // Cutie1 specificatii
    Cutie1.setLungime(6.0);
    Cutie1.setLatime(7.0);
    Cutie1.setInaltime(5.0);

    // Cutie2 specificatii
    Cutie2.setLungime(12.0);
    Cutie2.setLatime(13.0);
    Cutie2.setInaltime(10.0);

    // volum cutie1
    volum = Cutie1.getVolum();
    cout << "Volum Cutie1 : " << volum << endl;

    // volum cutie2
    volum = Cutie2.getVolum();
    cout << "Volum Cutie2 : " << volum << endl;

    // Adunare celor doua obiecte cutie:
    Cutie3 = Cutie1 + Cutie2;

    // volum Cutie3
    volum = Cutie3.getVolum();
    cout << "Volum Cutie3 : " << volum << endl;

    return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat:

```
Volum Cutie1 : 210

Volum Cutie2 : 1560

Volum Cutie3 : 5400
```

III.I.III Supraîncărcarea operatorilor relaționali

Limbajul C++ are un număr de operatori relaționali, inclusiv (, >, =, >=, ==, etc.) care pot fi utilizați pentru a compara tipurile de date încorporate C++.
Oricare dintre acești operatori, care pot fi utilizați pentru a compara obiectele unei clase, poate fi supraîncărcat.
Următorul exemplu arată cum un operator poate fi supraîncărcat și cum puteți supraîncărca operatori relaționali suplimentari într-un mod similar. *pentru accesarea codului consultați Anexa 14.

```cpp
#include <iostream>
using namespace std;

class Distanta {
private:
    int centimetri;
    int metri;
public:
    // constructori
    Distanta() {
        centimetri = 0;
        metri = 0;
    }
    Distanta(int f, int i) {
        centimetri = f;
        metri = i;
    }
    // afisarea distantelor
    void displayDistance() {
        cout << "Centimetri: " << centimetri << " Metri:" << metri << endl;
    }
    // supraincarcarea operatorului minus (-)
    Distanta operator- () {
        centimetri = -centimetri;
        metri = -metri;
        return Distanta(centimetri, metri);
    }
    // overloaded < operator
    bool operator <(const Distanta& d) {
        if (centimetri < d.centimetri) {
            return true;
        }
        if (centimetri == d.centimetri && metri < d.metri) {
            return true;
        }
        return false;
    }
};
int main() {
    Distanta D1(11, 10), D2(5, 11);

    if (D1 < D2) {
        cout << "D1 este mai mic decat D2 " << endl;
    }
    else {
        cout << "D2 este mai mic decat D1 " << endl;
    }
    return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat:

```
D2 este mai mic decat D1
```

III.I.IV Supraîncărcarea operatorilor de intrare/ieșire

Folosind operatorul de extragere a fluxului >> și operatorul de inserare a fluxului <<, C++ poate introduce și scoate tipurile de date încorporate. Supraîncărcarea operatorilor de inserare și extracție a

fluxului vă permite să executați intrare și ieșire pentru tipuri definite de utilizator, cum ar fi obiectele. Deoarece va fi invocat fără a construi un obiect, este esențial să declarați funcția de supraîncărcare a operatorului un prieten al clasei. Următorul exemplu demonstrează cum să utilizați operatorul de extracție >> și operatorul de inserare. *pentru a accesa codul de mai jos consultați Anexa15.

```
#include <iostream>
using namespace std;

class Distanta {
private:
    int centimetri;
    int metri;

public:
    // constructorii clasei
    Distanta() {
        centimetri = 0;
        metri = 0;
    }
    Distanta(int f, int i) {
        centimetri = f;
        metri = i;
    }
    friend ostream &operator<<(ostream &output, const Distanta &D) {
        output << "Centimetri : " << D.centimetri << " metri : " << D.metri;
        return output;
    }

    friend istream &operator>>(istream  &input, Distanta &D) {
        input >> D.centimetri >> D.metri;
        return input;
    }
};

int main() {
    Distanta D1(11, 10), D2(5, 11), D3;

    cout << "Introduceti valorile pentru obiectul nou : " << endl;
    cin >> D3;
    cout << "Prima Distanta: " << D1 << endl;
    cout << "A doua distanta:" << D2 << endl;
    cout << "A treia distanta:" << D3 << endl;

    return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat.

```
Introduceti valorile pentru obiectul nou :
10
12
Prima Distanta: Centimetri : 11 metri : 10
A doua distanta:Centimetri : 5 metri : 11
A treia distanta:Centimetri : 10 metri : 12
```

III.I.V Supraîncărcarea operatorilor ++ și --

În C++, operatorii de creștere (++) și de decrementare (--) sunt doi operatori unari utili. Exemplele de mai jos arată cum operatorul increment (++) poate fi supraîncărcat atât pentru utilizarea prefixului, cât și a postfixului. Puteți supraîncărca operatorul într-un mod similar (--). *pentru accesare cod consultați Anexa 16.

```cpp
#include <iostream>
using namespace std;

class Timp {
private:
    int ore;              // 0 la 23
    int minute;           // 0 la 59

public:
    // constructorul clasei
    Timp() {
        ore = 0;
        minute = 0;
    }
    Timp(int h, int m) {
        ore = h;
        minute = m;
    }

    // metoda care afiseaza ora
    void afiseazaTimp() {
        cout << "Ore: " << ore << " Minute:" << minute << endl;
    }

    // supraincarcarea operatorului ++ prefix
    Timp operator++ () {
        ++minute;          // incrementarea obiectului
        if (minute >= 60) {
            ++ore;
            minute -= 60;
        }
        return Timp(ore, minute);
    }

```

```
34
35      // supraincarcarea operatorului ++ postfix
36      Timp operator++(int) {
37
38          // salvare valoare originala
39          Timp T(ore, minute);
40
41          // incrementare obiect
42          ++minute;
43
44          if (minute >= 60) {
45              ++ore;
46              minute -= 60;
47          }
48
49          // returnare valoare veche
50          return T;
51      }
52  };
53
54  int main() {
55      Timp T1(11, 59), T2(10, 40);
56
57      ++T1;                       // incrementare T1
58      T1.afiseazaTimp();           // afisare T1
59      ++T1;                       // incrementare T1
60      T1.afiseazaTimp();           // afisare T1
61
62      T2++;                       // incrementare T2
63      T2.afiseazaTimp();           // afisare T2
64      T2++;                       // incrementare T2
65      T2.afiseazaTimp();           // afisare T2
66      return 0;
67  }
68
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat.

```
Ore: 12 Minute:0
Ore: 12 Minute:1
Ore: 10 Minute:41
Ore: 10 Minute:42
```

III.I.VI Supraîncărcarea operatorilor de atribuire

Operatorul de atribuire (=) poate fi supraîncărcat în același mod în care o pot face alți operatori și poate fi folosit pentru a genera obiecte în același mod în care poate face constructorul de copiere.

Exemplul de mai jos demonstrează modul în care un operator de atribuire poate fi supraîncărcat. *pentru accesare cod consultați Anexa 17.

```
#include <iostream>
using namespace std;

class Distanta {
private:
    int centimetri;
    int metri;

public:
    // constructorii clasei
    Distanta() {
        centimetri = 0;
        metri = 0;
    }
    Distanta(int f, int i) {
        centimetri = f;
        metri = i;
    }
    void operator = (const Distanta &D) {
        centimetri = D.centimetri;
        metri = D.metri;
    }

    // metoda pentru afisare distanta
    void afisareDistanta() {
        cout << "Centimetrii: " << centimetri << " Metrii:" << metri << endl;
    }
};

int main() {
    Distanta D1(11, 10), D2(5, 11);

    cout << "Prima distanta: ";
    D1.afisareDistanta();
    cout << "A doua distanta: ";
    D2.afisareDistanta();

    // utilizarea/invocarea operatorului de atribuire
    D1 = D2;
    cout << "Prima distanta: ";
    D1.afisareDistanta();

    return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat:

```
Prima distanta: Centimetrii: 11 Metrii:10
A doua distanta: Centimetrii: 5 Metrii:11
Prima distanta: Centimetrii: 5 Metrii:11
```

III.I.VII Supraîncărcarea operatorului apel de funcție ()

Pentru obiectele de tip clasă, operatorul de apelare a funcției () poate fi supraîncărcat. Nu inventați o nouă modalitate de a invoca o funcție atunci când supraîncărcați (). În schimb, construiți o funcție de operator care poate lua orice număr de intrări.

Exemplul de mai jos demonstrează modul în care un operator de apel de funcție () poate fi supraîncărcat. *pentru accesare cod consultati Anexa 18.

```
#include <iostream>
using namespace std;

class Distanta {
private:
    int centimetri;
    int metri;

public:
    // constructor clasa
    Distanta() {
        centimetri = 0;
        metri = 0;
    }
    Distanta(int f, int i) {
        centimetri = f;
        metri = i;
    }

    // supraincarcarea operatorului de apelare functie
    Distanta operator()(int a, int b, int c) {
        Distanta D;

        // calculul distantei
        D.centimetri = a + c + 10;
        D.metri = b + c + 100;
        return D;
    }

    // metoda pentru afisarea distantei
    void afisareDistanta() {
        cout << "Centimetri: " << centimetri << " Metri:" << metri << endl;
    }
};

int main() {
    Distanta D1(11, 10), D2;

    cout << "Prima distanta: ";
    D1.afisareDistanta();

    D2 = D1(10, 10, 10); // apelare operator ()
    cout << "A doua distanta: ";
    D2.afisareDistanta();

    return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat.

```
Prima distanta: Centimetri: 11 Metri:10

A doua distanta: Centimetri: 30 Metri:120
```

Capitolul 4: Polimorfismul în C++

Polimorfismul se referă la faptul că ceva există sub mai multe forme. Polimorfismul apare de obicei atunci când există o ierarhie de clase care sunt conectate prin moștenire.
Polimorfismul în C++ se referă la faptul că, în funcție de tipul de obiect care apelează o funcție membru, se rulează o funcție diferită.
Luați în considerare următorul scenariu, în care o clasă de bază este derivată din două clase suplimentare. *pentru accesare cod consultați Anexa 19.

```cpp
#include <iostream>
using namespace std;

class Forma {
protected:
    int latime, inaltime;
public:
    Forma(int a = 0, int b = 0) {
        latime = a;
        inaltime = b;
    }
    int arie() {
        cout << "Arie clasa parinte: " << endl;
        return 0;
    }
};
class Dreptunghi : public Forma {
public:
    Dreptunghi(int a = 0, int b = 0) :Forma(a, b) { }

    int arie() {
        cout << "Arie clasa dreptunghi: " << endl;
        return (latime * inaltime);
    }
};
class Triunghi : public Forma {
public:
    Triunghi(int a = 0, int b = 0) :Forma(a, b) { }

    int arie() {
        cout << "Arie clasa triunghi: " << endl;
        return (latime * inaltime / 2);
    }
};
int main() {
    Forma *forma;
    Dreptunghi rec(10, 7);
    Triunghi  tri(10, 5);

    // stocare adresa dreptunghi
    forma = &rec;
    // apelare metoda arie din clasa dreptunghi
    forma->arie();

    // stocare adresa triunghi
    forma = &tri;
    // apelare metoda arie din clasa triunghi
    forma->arie();

    return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat:

```
Aria clasa parinte :

Aria clasa parinte :
```

Compilatorul setează apelul metodei area() o dată ca versiune definită în clasa de bază, ceea ce are ca rezultat o ieșire incorectă. Apelul de funcție este fixat înainte ca programul să fie executat, ceea ce este cunoscut sub numele de rezoluție statică a apelului de funcție sau legătura statică. Deoarece funcția area() este setată în timpul compilării programului, aceasta este cunoscută și sub denumirea de legare timpurie.
Dar acum, să facem o mică modificare în programul nostru și să adăugăm cuvântul cheie virtual înainte de declararea area() în clasa Forma, astfel încât să arate așa.

```
class Forma {
protected:
	int latime, inaltime;

public:
	Forma(int a = 0, int b = 0) {
		latime = a;
		inaltime = b;
	}
	virtual int arie() {
		cout << "Arie clasa parinte: " << endl;
		return 0;
	}
};
```

După această ușoară modificare, atunci când exemplul de cod anterior este compilat și executat, acesta produce următorul rezultat.

```
Aria clasa dreptunghi:
```

Aria clasa triunghi:

De data aceasta, compilatorul examinează mai degrabă conținutul pointerului decât tipul acestuia. Deoarece adresele obiectelor claselor tri și rec sunt păstrate în formă *, este invocată metoda area().
După cum puteți vedea, zona de funcții este implementată diferit în fiecare dintre clasele copil (). Aceasta este cea mai comună aplicație a polimorfismului. Aveți clase care au același nume de funcție și chiar aceleași argumente, dar implementări distincte.

I. Funcții Virtuale

O funcție virtuală este o funcție specificată cu termenul virtual într-o clasă de bază. Când definim o funcție virtuală într-o clasă de bază și o altă versiune într-o clasă derivată, îi spunem compilatorului că nu dorim legături statice pentru această funcție.
Ceea ce dorim este ca funcția să fie apelată pe baza tipului de obiect pentru care este apelată în orice punct dat al programului. Legătura dinamică, uneori cunoscută sub numele de legare tardivă, este un termen folosit pentru a descrie acest tip de activitate.

II. Funcții Pur Virtuale

Este posibil să doriți să includeți o funcție virtuală într-o clasă de bază, astfel încât să poată fi redefinită într-o clasă derivată pentru a se potrivi cu obiectele acelei clase, dar nu puteți da funcției o definiție semnificativă în clasa de bază.
Metoda virtuală area() din clasa de bază poate fi modificată în următoarele:

```
class Forma {
protected:
       int latime, inaltime;
```

```
public:
	Forma(int a = 0, int b = 0) {
		latime = a;
		inaltime = b;
	}

	// functie pur virtuala
	virtual int arie() = 0;
};
```

= 0 îi spune compilatorului că funcția nu are corp și de mai sus funcția virtuală va fi numită funcție virtuală pură.

Capitolul 5: Interfețe în C++ (Clase Abstracte)

O interfață explică comportamentul sau capacitățile unei clase C++ fără a se angaja la o implementare specifică a acelei clase.
Clasele abstracte sunt folosite pentru implementarea interfețelor C++. Aceste clase abstracte nu trebuie confundate cu abstracția datelor, care este o noțiune care separă detaliile implementării de datele conectate.
O clasă devine abstractă atunci când cel puțin una dintre funcțiile sale este declarată ca o funcție virtuală pură. Expresia „= 0” în declarația unei funcții virtuale pure este următoarea:

```
class Cutie {
public:
	// pure virtual function
	virtual double getVolum() = 0;

private:
	double lungime;		// Lungime cutie
	double latime;		// Latime cutie
	double inaltime;	// Inaltime cutie
};
```

Scopul unei clase abstracte (de exemplu, ABC) este de a crea o clasă de bază adecvată din care pot deriva clase suplimentare. Clasele abstracte sunt folosite exclusiv ca interfață cu utilizatorul și nu pot fi utilizate pentru a crea obiecte. O eroare de compilare apare atunci când un obiect dintr-o clasă abstractă este încercat să fie instanțiat. Ca urmare, dacă trebuie creată o subclasă ABC, aceasta trebuie să implementeze fiecare dintre funcțiile virtuale, ceea ce implică faptul că acceptă interfața ABC. Erorile de compilare apar atunci când o clasă derivată nu reușește să suprascrie o funcție virtuală pură înainte de a încerca să creeze instanțe ale acelei clase.

Clasele concrete sunt cele care pot fi utilizate pentru a instanția obiecte.

I. Exemplu de clase abstracte

Luați în considerare următorul exemplu în care clasa părinte oferă o interfață pentru clasa de bază pentru a implementa o funcție numită getArie(). *pentru accesare cod consultați Anexa 20.

```
#include <iostream>
using namespace std;

// Clasa de baza - clasa parinte
class Forma {
public:
    // functie pur virtuala
    virtual int getArie() = 0;
    void setareLatime(int w) {
        latime = w;
    }
    void setareInaltime(int h) {
        inaltime = h;
    }
protected:
    int latime;
    int inaltime;
};
// Clasa derivata - clasa copil
class Dreptunghi : public Forma {
public:
    int getArie() {
        return (latime * inaltime);
    }
};
class Triunghi : public Forma {
public:
    int getArie() {
        return (latime * inaltime) / 2;
    }
};
int main(void) {
    Dreptunghi drept;
    Triunghi  tri;

    drept.setareLatime(5);
    drept.setareInaltime(7);
    // afisare arie obiect
    cout << "Arie dreptunghi: " << drept.getArie() << endl;

    tri.setareLatime(5);
    tri.setareInaltime(7);
    // afisare arie obiect
    cout << "Arie triunghi: " << tri.getArie() << endl;

    return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat.

```
Arie dreptunghi: 35

Arie triunghi: 17
```

Puteți vedea cum o clasă abstractă a stabilit o interfață în termeni de getArie() și cum alte două clase au implementat aceeași funcție, dar au folosit o tehnică diferită pentru a calcula zona specifică formei.
O clasă de bază abstractă poate fi utilizată într-un sistem orientat pe obiecte pentru a oferi o interfață comună și uniformă pentru toate aplicațiile externe. Apoi, prin moștenirea din acea clasă de bază abstractă, sunt create clase derivate cu funcționalitate echivalentă.
Capacitățile aplicațiilor externe (adică, funcțiile publice) sunt date ca funcții virtuale pure în clasa de bază abstractă. Clasele derivate care corespund anumitor tipuri de aplicație oferă implementări ale acestor funcții virtuale pure.
Această arhitectură simplifică, de asemenea, adăugarea de aplicații suplimentare la un sistem odată ce acesta a fost definit.

Capitolul 6: Încapsularea datelor în C++

Următoarele două componente esențiale sunt prezente în toate programele C++:

- **Declarațiile de program (codul)** sunt părțile unui program care desfășoară acțiuni. Ele sunt denumite funcții.

- **Date program -** sunt informațiile programului care sunt influențate de funcțiile programului.

Încapsularea este o noțiune din programarea orientată pe obiecte care leagă împreună datele și funcțiile pe care le gestionează, păstrându-le ambele protejate de intervenția externă și de utilizare greșită. Noțiunea crucială OOP de ascundere a datelor sa născut din încapsularea datelor [7].
Abstractizarea datelor este o tehnică pentru expunerea numai a interfețelor și ascunderea detaliilor de implementare de utilizator, în timp ce încapsularea datelor este o strategie pentru împachetarea datelor și a funcțiilor care le utilizează.
Încapsularea și ascunderea datelor sunt acceptate în C++ prin utilizarea claselor, care sunt tipuri definite de utilizator. Am văzut anterior că o clasă poate avea membri care sunt secreți, protejați sau publici. Toate articolele declarate într-o clasă sunt private în mod implicit. De exemplu,

```
class Cutie {
public:
	double getVolum(void) {
		return lungime * latime * inaltime;
	}

private:
```

```
        double lungime;       // Lungime cutie
        double latime;        // Latime cutie
        double inaltime;      // Inaltime cutie
};
```

Variabilele lungime, lățime și înălțime sunt toate private. Aceasta implică faptul că acestea pot fi accesibile numai de către alți membri ai clasei Cutie și nu de către orice altă parte a software-ului dumneavoastră. Aceasta este o metodă de încapsulare.
Trebuie să definiți secțiuni ale unei clase după cuvântul cheie public pentru a le face publice (adică, disponibile pentru alte părți ale programului dvs.). Toate celelalte funcții din aplicația dvs. pot accesa orice variabile sau funcții declarate după specificatorul public.
Făcând o clasă prietenă cu alta, detaliile implementării sunt expuse și încapsularea se pierde. Situația ideală este să păstrați cât mai multe informații ale fiecărei clase ascunse de alte clase.

I. Exemplu de încapsulare a datelor

Încapsularea datelor și abstractizarea lor pot fi văzute în fiecare program C++ care implementează o clasă cu membri publici și privați. Luați în considerare următoarea secvența de cod.

```
#include <iostream>
using namespace std;

class Sumator {
public:
    // constructor
    Sumator(int i = 0) {
        total = i;
    }
    // interfata externa
    void adunareNumere(int number) {
        total += number;
    }
    // interfata externa
    int returnezaTotal() {
        return total;
    };
private:
    // date ascunse mediului extern
    int total;
};

int main() {
    Sumator a;

    a.adunareNumere(10);
    a.adunareNumere(20);
    a.adunareNumere(30);

    cout << "Total: " << a.returnezaTotal() << endl;
    return 0;
}
```

Când codul de mai sus este compilat și executat, acesta produce următorul rezultat:

```
Total 60
```

Se returnează totalul numerelor din clasa de mai sus. Membrii publici adunareNumere și returneazaTotal sunt interfețele externe ale clasei, iar utilizatorul trebuie să fie familiarizat cu ele pentru a le utiliza. Membrul private total este ascuns de restul lumii, dar este necesar pentru ca clasa să funcționeze corect. Cu excepția cazului

în care avem o nevoie specifică de a dezvălui membrii clasei, cei mai mulți dintre noi am învățat să le facem private în mod implicit. Aceasta este o încapsulare excelentă.

Acesta este cel mai frecvent aplicat membrilor de date, deși poate fi aplicat oricărui membru, inclusiv funcțiilor virtuale.

ANEXE

Anexa 1

```
#include <iostream>

using namespace std;

class Cutie {

public:

        double lungime;   // Lungimea cutiei

        double latime;  // Latimea cutiei

        double inaltime;   // Inaltimea cutiei

};

main() {

        Cutie Cutie1;        // Declarare Cutie 1 de tip Cutie

        Cutie Cutie2;        // Declarare Cutie 2 de tip Cutie

        double volumCutie = 0.0;     // Stocare colum cutie in
variabila volumCutie


        Cutie1.inaltime = 5.0;

        Cutie1.lungime = 6.0;

        Cutie1.latime = 7.0;
```

```
        Cutie2.inaltime = 10.0;
        Cutie2.lungime = 12.0;
        Cutie2.latime = 13.0;


        // Calcul volum pentru cutie1
        volumCutie = Cutie1.inaltime * Cutie1.lungime *
Cutie1.latime;
        std::cout << "Volum pentru Cutie1 : " << volumCutie <<
std::endl;


        // Calcul volum pentru cutie2
        volumCutie = Cutie2.inaltime * Cutie2.lungime *
Cutie2.latime;
        std::cout << "Volum pentru Cutie2 : " << volumCutie <<
std::endl;
        return 0;
}
```

Anexa 2

```
#include <iostream>
using namespace std;
class Cutie {
  public:
    double lungime;        // lungimea cutiei
    double latime;       // latimea cutiei
    double inaltime;        // inaltimea cutiei

 // Prototip functii membru
    double getVolum(void);
    void setLungime( double len );
    void setLatime( double bre );
    void setInaltime( double hei );
};

// Member functions definitions
double Cutie::getVolum(void) {
```

```
   return lungime * inaltime * latime;
}
void Cutie::setLungime( double len ) {
   lungime = len;
}
void Cutie::setLatime( double bre ) {
   latime = bre;
}
void Cutie::setInaltime( double hei ) {
   inaltime = hei;
}

// Functia Main
int main() {
   Cutie Cutie1;              // Declarare Cutie1 de tip Cutie
   Cutie Cutie2;              // Declarare Cutie2 de tip Cutie
   double volum = 0.0;        // Declarare si initializare variabila volum

 // specificatii cutie1
```

```
    Cutie1.setLungime(6.0);
    Cutie1.setLatime(7.0);
    Cutie1.setInaltime(5.0);

   // specificatii cutie2
    Cutie2.setLungime(12.0);
    Cutie2.setLatime(13.0);
    Cutie2.setInaltime(10.0);

  // calculare volum pt cutie1
    volum = Cutie1.getVolum();
    cout << "Volumul Cutie1 : " << volum <<endl;
    // calculare volum pt cuti2
    volum = Cutie2.getVolum();
    cout << "Volum Cutie2 : " << volum <<endl;
    return 0;
}
```

Anexa 3

```
#include <iostream>

using namespace std;

class Linie {
  public:
    double lungime;
    void setLungime( double len );
    double getLungime( void );
};

// definirea functiilor membru
double Linie::getLungime(void) {
  return lungime ;
}

void Linie::setLungime( double len) {
```

```
    lungime = len;
}

// functia Main a programului
int main() {
    Linie linie;

    // setare lungime linie
    linie.setLungime(6.0);
    cout << "Lungimea liniei este: " << linie.getLungime() <<endl;

    // setare lungime fara a utiliza functii membru
    linie.lungime = 10.0; // OK: deoarece lungime este declarata public
    cout << "Lungimea liniei este: " << linie.lungime <<endl;

    return 0;
}
```

Anexa 4

```
#include <iostream>

using namespace std;

class Linie {

public:
        int getLungime(void);
        Linie(int len);             // constructor simplu
        Linie(const Linie &obj);  // constructor copiere
        ~Linie();                   // destructor

private:
        int *ptr;
};
// functii membru
Linie::Linie(int len) {
```

```
        cout << "Constructor normal - alocare ptr" << endl;


        // alocare memorie pentru pointer;
        ptr = new int;
        *ptr = len;
}
Linie::Linie(const Linie &obj) {
        cout << "Constructor copiere - alocare ptr." << endl;
        ptr = new int;
        *ptr = *obj.ptr; // copiere valoare
}
Linie::~Linie(void) {
        cout << "Eliberare memorie!" << endl;
        delete ptr;
}
int Linie::getLungime(void) {
        return *ptr;
}
```

```
void display(Linie obj) {
	cout << "Lungime linie este: " << obj.getLungime() << endl;
}

int main() {
	Linie linie(10);

	display(linie);

	return 0;
}
```

Anexa 5

```
#include <iostream>

using namespace std;

class Cutie {
        double latime;

public:
        friend void printeazaLatime(Cutie box);
        void setLatime(double wid);
};

// Member function definition
void Cutie::setLatime(double wid) {
        latime = wid;
}
```

```
// Nota: printeazaLatime() nu este functie membru in nici o clasa.
void printeazaLatime(Cutie cutie) {
	/* Deoarece printeazaLatime() este un prieten al clasei Cutie, poate accesa direct
	orice membru al clasei Cutie */
	cout << "Latime cutie este: " << cutie.latime << endl;
}

int main() {
	Cutie cutie;

	// setare latime cutie fara functii membru
	cutie.setLatime(10.0);

	// utilizare functie prieten pentru afisarea latimii.
	printeazaLatime(cutie);

	return 0;
}
```

Anexa 6

```
#include <iostream>

using namespace std;

class Cutie {
public:
	// definire constructor
	Cutie(double l = 2.0, double b = 2.0, double h = 2.0) {
		cout << "Apelare Constructor." << endl;
		lungime = l;
		latime = b;
		inaltime = h;
	}
	double Volum() {
		return lungime * latime * inaltime;
	}
	int comparare(Cutie cutie) {
```

```
            return this->Volum() > cutie.Volum();
        }

private:
        double lungime;     // Lungime cutie
        double latime;    // Latime cutie
        double inaltime;     // Inaltime cutie
};

int main(void) {
        Cutie Cutie1(3.3, 1.2, 1.5);    // Declarare Cutie1
        Cutie Cutie2(8.5, 6.0, 2.0);    // Declarare Cutie2

        if (Cutie1.comparare(Cutie2)) {
            cout << "Cutie2 este mai mica decat Cutie1" << endl;
        }
        else {
            cout << "Cutie2 este egala sau mai mare decat Cutie1"
<< endl;
```

```
        }

        return 0;
}
```

Anexa 7

```
#include <iostream>

using namespace std;

class Cutie {
public:
	static int contor;

	// Definire constructor
	Cutie(double l = 2.0, double b = 2.0, double h = 2.0) {
		cout << "Constructor apelat." << endl;
		lungime = l;
		latime = b;
		inaltime = h;

		// se incrementeaza contor de fiecare data cand un
pbiect este creat
		contor++;
```

```
        }
        double Volum() {
                return lungime * latime * inaltime;
        }

private:
        double lungime;
        double latime;
        double inaltime;
};

// Initializarea membrului static din clasa Cutie
int Cutie::contor = 0;

int main(void) {
        Cutie Cutie1(3.3, 1.2, 1.5);    // Declarare cutie1
        Cutie Cutie2(8.5, 6.0, 2.0);    // Declarare cutie2
```

```
                                                    // Afisare
numar  total de obiecte.
        cout << "Totalul obiectelor este: " << Cutie::contor << endl;


        return 0;
}
```

Anexa 8

```
#include <iostream>
using namespace std;

class Cutie {
public:
	static int contor;
	// Definire constructor
	Cutie(double l = 2.0, double b = 2.0, double h = 2.0) {
		cout << "Constructor apelat." << endl;
		lungime = l;
		latime = b;
		inaltime = h;
		// se incrementeaza la crearea fiecarui obiect nou
		contor++;
	}
	double Volum() {
		return lungime * latime * inaltime;
```

```
        }
        static int getContor() {
                return contor;
        }
private:
        double lungime;
        double latime;
        double inaltime;
};
// Initializarea membrului static din clasa Cutie
int Cutie::contor = 0;


int main(void) {
        // Afisare valoare contor inainte de creare obiecte.
        cout << "Valoare initiala contor: " << Cutie::getContor() <<
endl;


        Cutie Cutie1(3.3, 1.2, 1.5);
        Cutie Cutie2(8.5, 6.0, 2.0);
```

```
                                                    // Afisare
total numar de obiecte create.

        cout << "Valoare contor finala: " << Cutie::getContor() <<
endl;


        return 0;

}
```

Anexa 9

```
#include <iostream>

using namespace std;

// Clasa de baza
class Forma {
public:
	void setLatime(int w) {
		latime = w;
	}
	void setInaltime(int h) {
		inaltime = h;
	}

protected:
	int latime;
	int inaltime;
```

```
};
// Clasa derivata
class Dreptunghi : public Forma {
public:
	int getArie() {
		return (latime * inaltime);
	}
};
int main(void) {
	Dreptunghi dreptunghi;

	dreptunghi.setLatime(5);
	dreptunghi.setInaltime(7);

	// se afiseaza aria obiectului.
	cout << "Aria este: " << dreptunghi.getArie() << endl;

	return 0;
```

Anexa 10

```
#include <iostream>
using namespace std;
// Clasa de baza 1
class Forma {
public:
	void setLatime(int w) {
		latime = w;
	}
	void setInaltime(int h) {
		inaltime = h;
	}
protected:
	int latime;
	int inaltime;
};
// Clasa de baza 2
class CostZugravit {
```

```
public:
        int getCost(int aria) {
                return aria * 70;
        }
};
// Clasa Derivata
class Dreptunghi : public Forma, public CostZugravit {
public:
        int getAria() {
                return (latime * inaltime);
        }
};
int main(void) {
        Dreptunghi dreptunghi;
        int area;

        dreptunghi.setLatime(5);
        dreptunghi.setInaltime(7);
        area = dreptunghi.getAria();
```

```
        // Afiseaza aria obiectului.
        cout << "Aria totala: " << dreptunghi.getAria() << endl;
        // Afiseaza costul total pentru zugravirea obiectului
        cout << "Cost total pentru zugravit: RON " <<
dreptunghi.getCost(area) << endl;


        return 0;
}
```

Anexa 11

```
#include <iostream>
using namespace std;

class PrintareDate {
public:
	void printare(int i) {
		cout << "Afisare valoare int: " << i << endl;
	}
	void printare(double  f) {
		cout << "Afisare valoare float: " << f << endl;
	}
	void printare(char* c) {
		cout << "Afisare valoare character: " << c << endl;
	}
};

int main(void) {
```

```
    PrintareDate pd;
    // Apelare metoda print cu paramentru de tip intreg
    pd.printare(5);
    // Apelare metoda print cu paramentru de tip float
    pd.printare(100.263);
    // Apelare metoda print cu paramentru de tip caracter
    pd.printare("Salut din C++");
    return 0;
}
```

Anexa 12

```
#include <iostream>
using namespace std;

class Cutie {
public:
	double getVolum(void) {
		return lungime * latime * inaltime;
	}
	void setLungime(double len) {
		lungime = len;
	}
	void setLatime(double bre) {
		latime = bre;
	}
	void setInaltime(double hei) {
		inaltime = hei;
	}
```

```
        // supraincarcarea operatorului + pentru a aduna doua obiecte de tipul Cutie.
        Cutie operator+(const Cutie& b) {
                Cutie cutie;
                cutie.lungime = this->lungime + b.lungime;
                cutie.latime = this->latime + b.latime;
                cutie.inaltime = this->inaltime + b.inaltime;
                return cutie;
        }

private:
        double lungime;      // Lungime cutie
        double latime;     // Latime cutie
        double inaltime;      // Inaltime cutie
};

int main() {
        Cutie Cutie1;               // Declarare Cutie1 de tip Cutie
```

```
Cutie Cutie2;              // Declarare Cutie2 de tip Cutie
Cutie Cutie3;              // Declarare Cutie3 de tip Cutie
double volum = 0.0;        // variabila stocare volum cutie

// Specificatii cutie1
Cutie1.setLungime(6.0);
Cutie1.setLatime(7.0);
Cutie1.setInaltime(5.0);

// Specificatii cutie2
Cutie2.setLungime(12.0);
Cutie2.setLatime(13.0);
Cutie2.setInaltime(10.0);

// volum cutie1
volum = Cutie1.getVolum();
cout << "Volum al Cutie1 : " << volum << endl;

// volum cutie2
```

```
    volum = Cutie2.getVolum();
    cout << "Volum al Cutie 2 : " << volum << endl;

    // Adunarea celor doua obiecte cutie:
    Cutie3 = Cutie1 + Cutie2;

    // volum cutie3
    volum = Cutie3.getVolum();
    cout << "Volum al Cutie 3 : " << volum << endl;

    return 0;
}
```

Anexa 13

```
#include <iostream>

using namespace std;


class Cutie {
        double lungime;
        double latime;
        double inaltime;
public:
        double getVolum(void) {
                return lungime * latime * inaltime;
        }
        void setLungime(double len) {
                lungime = len;
        }
        void setLatime(double bre) {
                latime = bre;
        }
        void setInaltime(double hei) {
```

```
            inaltime = hei;
        }
        // supraincarcare operator + pentru a aduna doua obiecte de
tip Cutie
        Cutie operator+(const Cutie& b) {
            Cutie cutie;
            cutie.lungime = this->lungime + b.lungime;
            cutie.latime = this->latime + b.latime;
            cutie.inaltime = this->inaltime + b.inaltime;
            return cutie;
        }
};
int main() {
        Cutie Cutie1;           // Declarare Cutie1 de tip Cutie
        Cutie Cutie2;           // Declarare Cutie2 de tipu Cutie
        Cutie Cutie3;           // Declarare Cutie3 de tipu Cutie
        double volum = 0.0;     // Stocare date despre volum in
aceasta variabila

         // Cutie1 specificatii
```

```
Cutie1.setLungime(6.0);
Cutie1.setLatime(7.0);
Cutie1.setInaltime(5.0);

// Cutie2 specificatii
Cutie2.setLungime(12.0);
Cutie2.setLatime(13.0);
Cutie2.setInaltime(10.0);

// volum cutie1
volum = Cutie1.getVolum();
cout << "Volum Cutie1 : " << volum << endl;

// volum cutie2
volum = Cutie2.getVolum();
cout << "Volum Cutie2 : " << volum << endl;

// Adunare cele doua obiecte cutie:
Cutie3 = Cutie1 + Cutie2;
```

```
        // volum Cutie3
        volum = Cutie3.getVolum();
        cout << "Volum Cutie3 : " << volum << endl;

        return 0;
}
```

Anexa 14

```
#include <iostream>
using namespace std;
class Distanta {
private:
	int centimetri;
	int metri;
public:
	// constructori
	Distanta() {
		centimetri = 0;
		metri = 0;
	}
	Distanta(int f, int i) {
		centimetri = f;
		metri = i;
	}
	// afisarea distantelor
```

```
        void displayDistance() {
                cout << "Centimetri: " << centimetri << " Metri:" <<
metri << endl;
        }
        // supraincarcarea operatorului minus (-)
        Distanta operator- () {
                centimetri = -centimetri;
                metri = -metri;
                return Distanta(centimetri, metri);
        }
        // overloaded < operator
        bool operator <(const Distanta& d) {
                if (centimetri < d.centimetri) {
                        return true;
                }
                if (centimetri == d.centimetri && metri < d.metri) {
                        return true;
                }
                return false;
```

```
        }};
int main() {
        Distanta D1(11, 10), D2(5, 11);
        if (D1 < D2) {
                cout << "D1 este mai mic decat D2 " << endl;
        }
        else {
                cout << "D2 este mai mic decat D1 " << endl;
        }
        return 0;
}
```

Anexa 15

```
#include <iostream>
using namespace std;

class Distanta {
private:
	int centimetri;
	int metri;

public:
	// constructorii clasei
	Distanta() {
		centimetri = 0;
		metri = 0;
	}
	Distanta(int f, int i) {
		centimetri = f;
		metri = i;
```

```
        }
        friend ostream &operator<<(ostream &output, const Distanta
&D) {
                output << "Centimetri : " << D.centimetri << " metri :
" << D.metri;
                return output;
        }

        friend istream &operator>>(istream  &input, Distanta &D) {
                input >> D.centimetri >> D.metri;
                return input;
        }
};

int main() {
        Distanta D1(11, 10), D2(5, 11), D3;

        cout << "Introduceti valorile pentru obiectul nou : " << endl;
        cin >> D3;
        cout << "Prima Distanta: " << D1 << endl;
```

```
        cout << "A doua distanta:" << D2 << endl;
        cout << "A treia distanta:" << D3 << endl;


        return 0;
}
```

Anexa 16

```
#include <iostream>

using namespace std;


class Timp {

private:

        int ore;            // 0 la 23

        int minute;         // 0 la 59


public:

        // constructorii clasei

        Timp() {

                ore = 0;

                minute = 0;

        }

        Timp(int h, int m) {

                ore = h;

                minute = m;

        }
```

```
	// metoda care afiseaza ora
	void afiseazaTimp() {
		cout << "Ore: " << ore << " Minute:" << minute <<
endl;
	}

	// supraincarcarea operatorului ++ prefix
	Timp operator++ () {
		++minute;		// incrementare obiectului
		if (minute >= 60) {
			++ore;
			minute -= 60;
		}
		return Timp(ore, minute);
	}

	// supraincarcarea operatorului ++ postfix
	Timp operator++(int) {
```

```
                // salvare valoare originala
                Timp T(ore, minute);

                // incrementare obiect
                ++minute;

                if (minute >= 60) {
                        ++ore;
                        minute -= 60;
                }

                // returnare valoare veche
                return T;
        }
};

int main() {
        Timp T1(11, 59), T2(10, 40);
```

```
        ++T1;                   // incrementare T1
        T1.afiseazaTimp();      // afisare T1
        ++T1;                   // incrementare T1
        T1.afiseazaTimp();      // afisare T1

        T2++;                   // incrementare T2
        T2.afiseazaTimp();      // afisare T2
        T2++;                   // incrementare T2
        T2.afiseazaTimp();      // afisare T2
        return 0;
}
```

Anexa 17

```
#include <iostream>
using namespace std;

class Distanta {
private:
	int centimetri;
	int metri;

public:
	// constructorii clasei
	Distanta() {
		centimetri = 0;
		metri = 0;
	}
	Distanta(int f, int i) {
		centimetri = f;
		metri = i;
```

```
        }
        void operator = (const Distanta &D) {
                centimetri = D.centimetri;
                metri = D.metri;
        }

        // metoda pentru afisare distanta
        void afisareDistanta() {
                cout << "Centimetrii: " << centimetri << " Metrii:" <<
metri << endl;
        }
};

int main() {
        Distanta D1(11, 10), D2(5, 11);

        cout << "Prima distanta: ";
        D1.afisareDistanta();
        cout << "A doua distanta: ";
```

```
        D2.afisareDistanta();

        // utilizarea/invocarea operatorului de atribuire
        D1 = D2;
        cout << "Prima distanta: ";
        D1.afisareDistanta();

        return 0;
}
```

Anexa 18

```
#include <iostream>
using namespace std;

class Distanta {
private:
	int centimetri;
	int metri;

public:
	// constructorii clasei
	Distanta() {
		centimetri = 0;
		metri = 0;
	}
	Distanta(int f, int i) {
		centimetri = f;
		metri = i;
	}
```

```
	// supraincarcarea operatorului de apelare functie
	Distanta operator()(int a, int b, int c) {
		Distanta D;

		// calcule aleatorii
		D.centimetri = a + c + 10;
		D.metri = b + c + 100;
		return D;
	}

	// metoda pentru afisarea distantei
	void afisareDistanta() {
		cout << "Centimetri: " << centimetri << " Metri:" << metri << endl;
	}
};

int main() {
```

```
    Distanta D1(11, 10), D2;

    cout << "Prima distanta: ";
    D1.afisareDistanta();

    D2 = D1(10, 10, 10); // apelare operator ()
    cout << "A doua distanta: ";
    D2.afisareDistanta();

    return 0;
}
```

Anexa 19

```
#include <iostream>
using namespace std;

class Forma {
protected:
	int latime, inaltime;

public:
	Forma(int a = 0, int b = 0) {
		latime = a;
		inaltime = b;
	}
	int arie() {
		cout << "Arie clasa parinte: " << endl;
		return 0;
	}
};
class Dreptunghi : public Forma {
```

```
public:
        Dreptunghi(int a = 0, int b = 0) :Forma(a, b) { }

        int arie() {
                cout << "Arie clasa dreptunghi: " << endl;
                return (latime * inaltime);
        }
};

class Triunghi : public Forma {
public:
        Triunghi(int a = 0, int b = 0) :Forma(a, b) { }

        int arie() {
                cout << "Arie claas triunghi: " << endl;
                return (latime * inaltime / 2);
        }
};
```

```
int main() {
        Forma *forma;
        Dreptunghi rec(10, 7);
        Triunghi  tri(10, 5);

        // stocare adresa dreptunghi
        forma = &rec;

        // apelare metoda arie din clasa dreptunghi
        forma->arie();

        // stocare adresa triunghi
        forma = &tri;

        // apelare metoda arie din clasa triunghi
        forma->arie();

        return 0;
}
```

Anexa 20

```
#include <iostream>

using namespace std;

// Clasa de baza - clasa parinte
class Forma {
public:
	// functie pur virtuala
	virtual int getArie() = 0;
	void setareLatime(int w) {
		latime = w;
	}

	void setareInaltime(int h) {
		inaltime = h;
	}

protected:
```

```
        int latime;
        int inaltime;
};

// clasa derivata - clasa copil
class Dreptunghi : public Forma {
public:
        int getArie() {
                return (latime * inaltime);
        }
};

class Triunghi : public Forma {
public:
        int getArie() {
                return (latime * inaltime) / 2;
        }
};
```

```
int main(void) {
        Dreptunghi drept;
        Triunghi  tri;
        drept.setareLatime(5);
        drept.setareInaltime(7);
        // afisare arie obiect
        cout << "Arie dreptunghi: " << drept.getArie() << endl;
        tri.setareLatime(5);
        tri.setareInaltime(7);
        // afisare arie obiect
        cout << "Arie triunghi: " << tri.getArie() << endl;

        return 0;
}
```

Bibliografie

[1] Designing Object Oriented C++ Applications Using The Booch Method by Robert Cecil Martin, ISBN-13: 978-0132038379.

[2] Object-Oriented Programming in C++, Robert Lafore, ISBN 9780672323089.

[3] Effective Modern C++, SCOTT MEYERS, ISBN 1491903996.

[4] C++ Primer (5th Edition) 5th Edition, Stanley B, ISBN 9780133053036.

[5] Accelerated C++: Practical Programming, Andrew Koenig, ISBN 9781282652217.

[6] Effective Modern C++: 42 Specific Ways to Improve Your Use of C++,Scott Meyers ISBN978-1491903995.

[7] More Effective C++: 35 New Ways to Improve Your Programs and Designs (English Edition) 1st Edition, Scott Meyers, 978-0201633719.