3. Clase (III)

În acest capitol vom continua discuția despre clase şi abstractizarea datelor prezentând modul în care obiectele pot fi create şi sterse dinamic. De asemenea, vom vedea cum se lucrează cu obiectele constante şi funcțiile membre `const`, ce sunt „prietenii" unei clase, cum se folosesc datele şi funcțiile membre statice şi ce este pointerul `this`.

- *Obiecte constante şi funcții membre* ***`const`***

Principiul *restrângerii privilegiilor* (*principle of least privilege*) este unul fundamental în ingineria software. Să vedem cum se aplică el obiectelor.

Unele obiecte trebuie să fie modificabile, altele nu. Programatorul poate folosi cuvântul cheie `const` pentru a specifica faptul că un obiect nu poate fi modificat, ca în exemplul de mai jos în care obiectul `noon` este constant.

Exemplu

```
const Time noon(12, 0, 0);
```

Compilatoarele C++ permit funcțiilor membre să apeleze obiecte `const` doar dacă ele însele sunt declarate de asemenea `const`. Această regulă este valabilă inclusiv pentru funcții *get* care nu modifică obiectul. Funcțiile membre declarate `const` nu pot modifica obiectul.

O funcție const trebuie declarată în acest fel atât în prototip cât şi în definiția sa. Cuvântul cheie const trebuie plasat după lista de parametri ai funcției şi, atunci când este vorba despre o definiție, inainte de corpul funcției. De exemplu, funcția

```
int Time::getHour() const { return hour};
```

întoarce valoarea datei membre `hour` a unui obiect de tip `Time` şi, pentru că nu modifică valoarea obiectului este declarată `const`.

O problemă specială apare în cazul constructorilor şi al destructorilor. Acestora trebuie să li se permită să modifice obiectele pentru ca să poată face inițializări sau operații de ştergere.

Declararea obiectelor const accentuează pricipiul restrângerii privilegiilor, iar modificările incorecte ale obiectelor sunt detectate chiar din faza de compilare. Folosirea obiectelor, a funcțiilor membre şi a datelor membre `const` este crucială pentru o proiectare corectă a claselor şi a programelor, dar şi pentru codare. Declararea variabilelor şi obiectelor constante nu este utilă doar din punct de vedere al ingineriei software. Compilatoarele care se folosesc azi fac optimizări sofisticate asupra constantelor care nu pot fi făcute asupra variabilelor, îmbunătățind performanțele generale ale aplicațiilor.

Programul de mai jos este o extensie a clasei `Time` prezentată în capitolele precedente şi exemplifică folosirea ei pentru obiecte constante, neconstante şi a funcțiilor membre `const`. Compilând programul, veți observa că în funcția `main` sunt două erori care apar din apelul unor funcții `non-const` pentru obiecte `const`.

Exemplu

time5.h

```
#ifndef TIME5_H
#define TIME5_H
class Time
{
  public:
```

```
    Time(int = 0, int = 0, int = 0); //constructor

    //functii set
    void setTime(int, int, int);
    void setHour(int);   //seteaza hour
    void setMinute(int); //seteaza minute
    void setSecond(int); //seteaza second

    //functii get declarate const
    int getHour() const;   //intoarce hour
    int getMinute() const; //intoarce minute
    int getSecond() const; //intoarce second

    //functii print - ambele ar trebui declarate const
    void printShort() const; //tiparire in format scurt
    void printLong ();       //tiparire in format lung

  private:
    int hour;  //0-23
    int minute;  //0-59
    int second;  //0-59
};
#endif
```

time5.cpp

```
#include <iostream>
using std::cout;

#include "time5.h"

Time::Time(int hr, int min, int sec)
{
  setTime(hr, min, sec);
}

void Time::setTime(int h, int m, int s)
{
  setHour(h);
  setMinute(m);
  setSecond(s);
}

void Time::setHour(int h)
{
  hour = (h >= 0 && h < 24) ? h : 0;
}

void Time::setMinute(int m)
{
  minute = (m >= 0 && m < 60) ? m : 0;
}
```

```
void Time::setSecond(int s)
{
  second = (s >= 0 && s < 60) ? s : 0;
}

int Time::getHour() const { return hour; }
int Time::getMinute() const { return minute; }
int Time::getSecond() const { return second; }

void Time::printShort() const
{
  cout << (hour < 10 ? "0" : "") << hour << ":"
       << (minute < 10 ? "0" : "") << minute;
}

void Time::printLong()
{
  cout << ((hour == 0 || hour == 12) ? 12 : hour % 12)
       << ":" << (minute < 10 ? "0" : "") << minute
       << ":" << (second < 10 ? "0" : "") << second
       << (hour < 12 ? " AM" : " PM");
}
```

test_time5.cpp

```
#include "time5.h"

int main()
{
  Time wakeUp(6, 45, 0);     //obiect non-const
  const Time noon(12, 0, 0); //obiect constant

                       //FUNCTIE MEMBRA  OBIECT
  wakeUp.setHour(18); //non-const       non-const

  //EROARE:
  noon.setHour(12);   //non-const       const

  wakeUp.getHour();   //const           non-const
  noon.getMinute();   //const           const
  noon.printShort();  //const           const

  //EROARE:
  noon.printLong();   //non-const       const

  return 0;
}
```

Obiectul `wakeUp` este neconstant, iar `noon` este constant. Programul apelează funcțiile membre neconstante `noon.setHour(18)` şi `noon.printLong()` pentru obiectul `noon` care este constant, iar acest lucrul este semnalat de compilator care generează eroare.

Aşa cum am mai spus, constructorul nu poate fi `const`, însă el poate fi apelat pentru obiecte `const`, aşa cum se întâmplă în cazul obiectului `noon`. Să remarcăm

faptul că obiectele const nu pot fi modificate prin asignare, de aceea ele trebuie inițializate. Atunci când o dată membră este declarată const, definiția constructorului clasei trebuie modificată prin adăugarea unei *liste de inițializare a membrilor*. Vom ilustra inițializarea datelor membre const prin exemplul următor în care declarăm clasa `Increment` cu două date membre dintre care una este constantă.

Exemplu

test_const_data_member.cpp

```
#include <iostream>
using std::cout;
using std::endl;

class Increment
{
  public:
    Increment(int c = 0, int i = 1);
    void addIncrement(){ count += increment; }
    void print();
  private:
    int count;
    const int increment; //data membra const
};

//constructorul clasei Increment
Increment::Increment(int c, int i)
    : increment(i)
    //initializarea datei membre const increment
{
  count =c;
}

void Increment::print()
{
  cout << "count = " << count
       << ", increment = " << increment << endl;
}

int main()
{
  Increment value(10, 5);

  cout << "Inainte de incrementare: ";
  value.print();

  for(int j = 0; j < 3; j++)
  {
    value.addIncrement();
    cout << "Dupa increment " << j+1 << ": ";
    value.print();
  }
  return 0;
}
```

Rulând programul obținem următorul rezultat:

```
Inainte de incrementare: count = 10, increment = 5
Dupa increment 1: count = 15, increment = 5
Dupa increment 2: count = 20, increment = 5
Dupa increment 3: count = 25, increment = 5
```

Prin notația `: increment(i)` precizăm că data membră `increment` va fi inițializată cu valoarea `i`. Dacă sunt necesare mai multe inițializări, acestea sunt separate prin virgulă şi atunci avem o listă de inițializare.

Toate datele membre ale unei clase *pot* fi inițializate prin lista de inițializare a membrilor, însă datele membre `const` *trebuie* inițializate în felul acesta. Vom vedea tot în acest capitol că obiectele membre ale unei clase trebuie inițializate prin această metodă. Este vorba despre clasele care conțin ca date membre obiecte din alte clase. În capitolul următor vom vedea că, în cazul claselor derivate, partea care provine din clasa de bază trebuie inițializată tot prin lista de inițializare.

O variantă greşită a definiției constructorului clasei `Increment`, prin care data membră `increment` nu este inițializată prin lista de inițializare ci prin asignare, este următoarea:

```
//EROARE
Increment::Increment(int c, int i)
{
  count =c;
  increment = i; //nu se poate modifica valoarea unei date
                 //membre const
}
```

- *Compunerea claselor: Obiecte ca membri ai claselor*

Dacă am scrie un program pentru un ceas cu alarmă, un obiect al clasei AlarmClock ar trebui să „ştie” când trebuie să declanşeze alarma. O idee de proiectare a acestei clase ar fi includerea unui obiect Time ca membru al său. Acesta este un exemplu de *compunere a claselor*. O clasă poate conține ca membri obiecte ale altor clase.

Vom descrie conceptul de compunere a claselor printr-un exemplu în care clasa `Employee` conține două obiecte tip `Date`. Obiectele clasei `Employee` păstrează următoarele informații despre angajații unei firme: numele, prenumele, data naşterii şi data angajării. Datele membre `birthDate` şi `hireDate` sunt obiecte const ale clasei `Date` şi conțin, la rândul lor, datele membre `month`, `day` şi `year`. Vom impementa o variantă extinsă a clasei `Date` din capitolul anterior.

Exemplu

date1.h

```
#ifndef DATE1_H
#define DATE1_H
class Date
{
  public:
    //constructor implicit
    Date(int = 1, int = 1, int = 1990);
    void print() const;
    ~Date();
  private:
    int day;  //1-12
```

```
    int month;//1-31
    int year;
    //functie utilitara de testare a
    //corectitudinii zilei pentru month si year
    int checkDay(int);
};
#endif
```

date1.cpp

```
#include <iostream>
using std::cout;
using std::endl;
#include "date1.h"
//constructor fara verificarea valorilor
Date::Date(int d, int m, int y)
{
  if(m > 0 && m <= 12)
    month = m;
  else {
    month = 1;
    cout << "Luna " << m << " incorecta. "
         << "Valoarea implicita este 1.\n";
  }
  year = y;
  day = checkDay(d); //valideaza ziua
  cout << "Constructorul obiectului de tip Date pentru ";
  print();
  cout << endl;
}

//Tipareste data in forma zi-luna-an
void Date::print() const
  { cout << day << '-' << month << '-' << year; }

//Destructorul folosit pentru confirmarea stergerii
obiectului
Date::~Date()
{
  cout << "Destructorul obiectului de tip Date pentru ";
  print();
  cout << endl;
}

int Date::checkDay( int testDay )
{
  static const int daysPerMonth[13] =
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  if( testDay > 0 && testDay <= daysPerMonth[month])
    return testDay;

  //Februarie: test pentru an bisect
  if(month == 2 && testDay == 29 &&
```

```
      (year % 400 == 0 ||
      (year % 4 == 0 && year % 100 != 0)))
    return testDay;

  cout << "Ziua " << testDay << " incorecta. "
       << "Valoarea implicita este 1.\n";
  return 1;
}
```

employee1.h

```
#ifndef EMPLOYEE1_H
#define EMPLOYEE1_H
#include"date1.h"
class Employee
{
  public:
    Employee(char*, char*, int, int, int, int, int, int);
    void print() const;
    ~Employee();//destructor folosit la confirmarea ordinii
                //in care sunt sterse obiectele
  private:
    char firstName[25];
    char lastName[25];
    const Date birthDate;
    const Date hireDate;
};
#endif
```

employee1.cpp

```
#include <iostream>
using std::cout;
using std::endl;

#include <cstring>
#include "employee1.h"
#include "date1.h"

Employee::Employee(char* fname, char*lname,
           int bday, int bmonth, int byear,
           int hday, int hmonth, int hyear)
  : birthDate(bday, bmonth, byear),
    hireDate(hday, hmonth, hyear){
  //copiaza fname in firstName
  //verificand daca lungimea corespunde
  int length = strlen(fname);
  length = (length < 25 ? length : 24);
  strncpy( firstName, fname, length);
  firstName[length] = '\0';

  //copiaza lname in lastName
  //verificand daca lungimea corespunde
  length = strlen(lname);
  length = (length < 25 ? length : 24);
```

```
  strncpy( lastName, lname, length);
  lastName[length] = '\0';

  cout << "Constructorul obiectului Employee: "
       << firstName << ' ' << lastName << endl;
}

void Employee::print() const
{
  cout << lastName << ", " << firstName << "\nAngajat: ";
  hireDate.print();
  cout << "  Data nasterii: ";
  birthDate.print();
  cout << endl;
}

//Destructorul folosit pentru
//confirmarea stergerii obiectului
Employee::~Employee()
{
  cout << "Destructorul obiectului de tip Employee: "
       << lastName << ", " << firstName << endl;
}
```

test_composition.cpp

```
#include <iostream>
using std::cout;
using std::endl;

#include "employee1.h"

int main()
{
  Employee e("Bob", "Jones", 24, 7, 1949, 12, 3, 1988);

  cout << '\n';
  e.print();

  cout << "\nTesteaza constructorul Date "
       << "pentru valori incorecte:\n";
  Date d(35, 14, 1994);
  cout << endl;

  return 0;
}
```

Rulând acest program obținem următorul rezultat:

```
Constructorul obiectului de tip Date pentru 24-7-1949
Constructorul obiectului de tip Date pentru 12-3-1988
Constructorul obiectului Employee: Bob Jones

Jones, Bob
```

```
Angajat: 12-3-1988  Data nasterii: 24-7-1949

Testeaza constructorul Date pentru valori incorecte:
Luna 14 incorecta. Valoarea implicita este 1.
Ziua 35 incorecta. Valoarea implicita este 1.
Constructorul obiectului de tip Date pentru 1-1-1994

Destructorul obiectului de tip Date pentru 1-1-1994
Destructorul obiectului de tip Employee: Jones, Bob
Destructorul obiectului de tip Date pentru 12-3-1988
Destructorul obiectului de tip Date pentru 24-7-1949
```

Headerul constructorului clasei `Employee` din fişierul `employee1.cpp` conține şi lista de inițializare a obiectelor membre `birthDate` şi `hireDate` de tip `Date`:

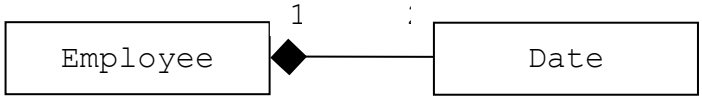
```
Employee::Employee(char* fname, char*lname,
                   int bday, int bmonth, int byear,
                   int hday, int hmonth, int hyear)
  : birthDate(bday, bmonth, byear),
    hireDate(hday, hmonth, hyear)
```

Constructorul are opt parametri. Lista de parametri este urmată de semnul `:` şi de lista de inițializare a membrilor clasei. Inițializarea membrilor constă din numele acestora, în cazul nostru `birthDate` şi `hireDate`, şi se argumentele care sunt transmise constructorilor celor două obiecte de tip `Date`. Argumentele `bday`, `bmonth` şi `byear` sunt transmise constructorului obiectului `birthDate`, iar Argumentele `hday`, `hmonth` şi h`year` sunt transmise constructorului obiectului `hireDate`.

Un obiect membru al unei clase nu trebuie inițializat explicit prin lista de inițializare. Dacă nu se face o astfel de inițializare pentru unul dintre obiectele membre, pentru acesta de apelează automat contructorul implicit al clasei sale care il va inițializa cu valori implicite. Lipsa constructorului implicit al clasei obiectului membru este semnalată de compilator ca eroare. Dacă în exemplul de mai sus nu am fi inițialitat obiectul `birthDate` prin lista de inițializare a constructorului clasei `Employee`, pentru inițializarea sa ar fi fost apelat automat constructorul cu valori implicite al clasei `Date`.

Obiectele membre ale unei clase sunt construite înaintea obiectului în care sunt incluse.

UML permite modelarea asocierii şi compunerii claselor. O metodă prin care clasele pot fi relaționate este cea de asociere. Pentru simplificarea diagramelor, UML permite truncherea simbolul de reprezentare a unei clase păstrând doar secțiunea care conține numele clasei. Diagrama de mai jos reprezintă compunerea claselor `Date` şi `Employee`.



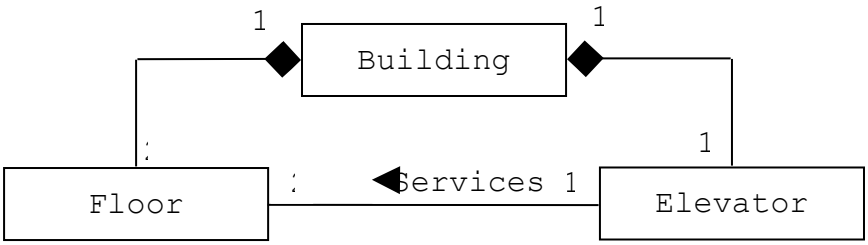
Compunerea este o relație tip parte/întreg. În exemplul nostru, clasa `Employee` conține obiecte ale clasei `Date`. Clasa `Employee` este întregul, iar obiectele clasei `Date` sunt părți ale acesteia.

Relația de asociere dintre cele două clase este reprezentată printr-o linie. Rombul negrul arată că asocierea este o relație de compunere. El este ataşat clasei

în compunerea căreia intră obiecte din clasa aflată la cealaltă extremitate a liniei. Valorile de multiplicitate arată câte obiecte ale claselor participă la relație. Conform diagramei de mai sus, două obiecte ale clasei `Date` fac parte din clasa `Employee`. Aceasta este o relație *unu-la-doi*. În diagramele UML se pot folosi următoarele valori de multiplicitate:

Simbol	**Semnificație**
0	Niciunul
1	Unu
m	O valoare întreagă
0..1	Zero sau unu
m..n	Cel puțin *m*, dar nu mai multe de *n*
*	Orice întreg nenegativ
0..*	Zero sau mai multe
1..*	Unu sau mai multe

Un alt exemplu este cel al claselor `Building`, `Floor` şi `Elevator` care fac parte dintr-o aplicație care dirijează funcționarea unui lift.



În această diagramă, două obiecte `Floor` participă în compunere cu un obiect din clasa `Building`. Pe de altă parte, un obiect din clasa `Elevator` face parte din clasa `Building`. Clasele `Floor` şi `Elevator` sunt într-o relație de asociere. Unei relații i se poate da un nume. De exemplu, cuvântul „Services” de deasupra liniei care conectează clasele `Floor` şi `Elevator` indică numele asocierii. Săgeata arată direcția asocierii. Un obiect al clasei `Elevator` deserveşte două obiecte ale clasei `Floor`, fiind o asociere *unu-la-doi*.

Diferența dintre asocierea simplă şi compunere este dată de perioada de existență a obiectelor implicate. În cazul compunerii, obiectele fac parte integrantă din clasa care reprezintă întregul şi perioadele lor de existență coincid. Întregul este responsabil de crearea şi distrugerea obiectelor care intră în compunerea lui. Atunci când avem de a face cu o simplă asociere, obiectele parte şi întreg pot avea perioade de existență diferite. În C++, această asociere poate fi modelată prin includerea în clasa întreg a unor referințe la obiecte din alte clase. O altă metodă de modelare a asocierii simple este folosirea de către funcțiile membre ale clasei întreg de parametri de tipul claselor parte.

- *Funcții `friend` şi clase `friend`*

O funcție `friend` a unei clase se defineşte în afara domeniului clasei, dar are drept de acces la membrii `private` (şi `protected`, aşa cum vom vedea atunci când vom prezenta moştenirea) ai clasei. O funcție sau o clasă întreagă pot fi declarate `friend` pentru o alta clasă.

În această secțiune vom prezenta un exemplu mecanic, prin care ilustrăm strict sintaxa şi modul de acces la membri clasei. Într-un alt capitol al cursului vom folosi funcțiile friend la supraîncărcarea unor operatori pentru a putea fi folosiți de obiectele

unei clase sau la crearea claselor iterator. Obiectele unei clase iterator se folosesc pentru a selecta elemente succesive sau pentru a efectua operații asupra elementelor dintr-un obiect container. Obiectele claselor container păstrează colecții de elemente. Funcțiile `friend` se folosesc în mod obişnuit atunci când anumite operații nu pot fi implementate prin funcții membre, ca în cazul supraîncărcării operatorilor.

Declararea unei funcții ca „prietenă” a unei clase se face prin inserarea prototipului său precedat de cuvântul cheie `friend` în clasa respectivă. Relația de „prietenie” între două clase, de exemplu `ClassOne` şi `ClassTwo`, se exprimă prin plasarea declarației

```
friend class ClassTwo;
```

în definiția clasei `ClassOne`.

Programul următor demonstrează modul în care se declară şi se foloseşte funcția globală `setX` care este `friend` pentru clasa `Count`. Această funcție setează data membră privată `x` din clasa `Count`.

Exemplu

test_friend.h

```
#include <iostream>
using std::cout;
using std::endl;

class Count
{
  friend void setX(Count&, int);//declaratie friend
  public:
    Count(){ x = 0; } // constructor
    void print() const { cout << x << endl; }//iesire
  private:
    int x; //data membra
};

//Poate sa modifice data membra privata a clasei Count
//pentru ca setX este declarata functie friend a clasei Count
void setX( Count& c, int val )
{
  c.x = val;
}

int main()
{
  Count counter;

  cout << "counter.x dupa instantiere: ";
  counter.print();
  cout << "counter.x dupa apelul functiei friend setX: ";
  setX(counter, 8);
  counter.print();

  return 0;
}
```

Rezultatul rulării acestui program este:

```
counter.x dupa instantiere: 0
counter.x dupa apelul functiei friend setX: 8
```

În acest exemplu, funcția `setX` nu este membră a clasei `Count`. Acesta este motivul pentru care atunci când invocăm `setX` pentru obiectul `counter` folosim instrucțiunea

```
setX(counter, 8);
```

care îl ia pe counter ca argument, neputând să folosim un apel ca în

```
counter.setX(8);
```

Dacă definiția clasei Count nu ar conține declarația friend a funcției setX, astfel:

```
//Clasa Count modificata: nu contine declaratia friend
//a functiei setX
class Count
{
  public:
    Count(){ x = 0; } // constructor
    void print() const { cout << x << endl; }//iesire
  private:
    int x; //data membra
};
```

funcția `setX` nu ar avea acces la data membră privată `x` şi compilatorul ar genera eroare, apelul `setX(counter, 8);` fiind considerat incorect.

Cu toate că specificatorii de acces `public`, `protected` şi `private` nu au nicio influență asupra declarațiilor `friend`, se preferă ca aceste declarații să fie plasate imediat după headerul clasei.

Există părerea în comunitatea dezvoltatorilor de aplicații orientate pe obiecte că declarațiile `friend` sunt în contradicție cu ideea de ascundere a informației şi că slăbeşte valoarea abordării orientării pe obiecte.

- *Pointerul `this`*

Fiecare obiect are acces la propria adresă de memorie prin intermediul pointerului numit `this`. Pointerul `this` al unui obiect nu este parte a obiectului, el nefiind reflectat de rezultatul unei operații sizeof asupra sa. Pe de altă parte, `this` este introdus de compilator ca prim argument în fiecare apel al funcțiilor nestatice realizat de obiect.

Pointerul `this` este folosit implicit pentru a referi datele membre şi funcțiile membre ale unui obiect. El poate fi folosit şi explicit. Tipul pointerului this depinde de tipul obiectului şi de caracterul const sau non-const al funcției membre apelate de obiect. Într-o funcție membră non-`const` a clasei `Employee`, pointerul `this` are tipul de dată `Employee* const` (pointer constant la un obiect `Employee`). Într-o funcție membră constantă a clasei Employee, pointerul this are tipul de dată `const Employee* const` (pointer constant la un obiect Employee constant).

Exemplul următor prezintă modul în care se poate folosi pointerul `this` în mod explicit. Fiecare funcție membră non-`static` are acces la pointerul `this` al obiectului pentru care este invocată funcția membră. Funcția `print` din clasa `Test` tipăreşte valoarea datei membre private `x` prin intermediul pointerului `this`.

Exemplu

test_this.h

```
#include <iostream>
```

```
using std::cout;
using std::endl;

class Test
{
  public:
    Test( int = 0); // constructor implicit
    void print() const;
  private:
    int x;
};

Test::Test( int a ) { x = a; }

void Test::print() const
{
  cout << "         x = " << x
       << "\n  this->x = " << this->x
       << "\n(*this).x = " << (*this).x
       << endl;
}

int main()
{
  Test testObject(12);
  testObject.print();

  return 0;
}
```

La rularea acestui program se obține următorul rezultat:

```
         x = 12
  this->x = 12
(*this).x = 12
```

Valoarea datei membre `x` din obiectul `testObject` este tipărită în trei moduri. Mai întâi, aceasta este afişată direct. Apoi funcția `print` foloseşte două notații diferite pentru a accesa valoarea lui `x` prin pointerul `this`: operatorul săgeată `->` pentru pointerul `this` la obiect şi operatorul punct `.` pentru pointerul `this` dereferențiat.

Folosirea parantezelor care încadrează pointerul dereferențiat `*this` este obligatorie pentru că operatorul `.` are precedență mai mare decât `*`. Fără paranteze, expresia `*this.x` ar fi evaluată ca şi cum parantezele ar fi plasate după operatorul de dereferențiere, `*(this.x)` care este o eroare de sintaxă pentru că operatorul `.` nu poate fi folosit pentru pointeri.

- ***Alocarea dinamică a memoriei cu operatorii `new` şi `delete`***

Alocarea dinamică a memoriei înseamnă posibilitatea ca programul să obțină mai mult spațiu de memorie în timpul execuției.

Operatorii `new` şi `delete` sunt folosiți în limbajul C++ pentru alocarea şi dealocarea dinamică a memoriei. Ei înlocuiesc apelurile de funcții `malloc` şi `free` care erau folosite în limbajul C.

Considerăm declarația

```
TypeName *typeNamePtr;
```

Alocarea dinamică a memoriei se poate face în limbajul C prin instrucțiunea

```
typeNamePtr = malloc(sizeof(TypeName));
```

însă funcția `malloc` nu oferă nicio metodă de inițializare a blocului de memorie alocat. În C++, putem scrie mai simplu:

```
typeNamePtr = new TypeName;
```

Prin acestă instrucțiune operatorul `new` creează automat în memorie un obiect cu dimensiunea dată de tipul de dată invocat, apelează constructorul pentru noul obiect şi returnează un pointer de tipul obiectului. Dacă `new` nu găseşte suficient spațiu pentru obiect, returnează valoarea `0` dacă folosim un compilator anterior standardului ANSI/ISO C++ sau, pentru compilatoarele care respectă acest standard, o excepție care poate fi tratată în codul client.

Pentru a şterge din memorie obiectele alocate dinamic şi a elibera memoria se foloseşte operatorul `delete` dacă alocarea a fost făcută prin operatorul `new` sau funcția `free` dacă am folosit `malloc` şi putem scrie:

```
delete typeNamePtr;
```

Operatorul `delete` apelează destructorul clasei din care face parte obiectul a cărul zonă de memorie urmează să fie eliberată, dacă în clasă acesta este declarat. Combinarea alocării memoriei printr-o metodă cu eliberarea acesteia prin cealaltă metodă este o eroare de logică a programului. Memoria alocată cu `malloc` nu este eliberată de operatorul `delete`, iar memoria alocata cu `new` nu este eliberată de funcția `free`.

Limbajul C++ permite transmiterea unor valori pentru inițializarea obiectelor create dinamic:

```
double *thingPtr = new double(3.1415927);
```

care inițializează noul obiect de tip `double` cu valoarea `3.1415927`.

Tablourile pot fi alocate dinamic folosind modelul următor:

```
int *arrayPtr = new int[10];
```

iar eliberarea memoriei se face prin instrucțiunea

```
delete [] arrayPtr;
```

Este recomandată folosirea parantezelor drepte pentru eliberarea spațiului ocupat de tablouri pentru a evita erorile care pot apărea la rularea programului.

- *Membrii `static` ai claselor*

Fiecare obiect al unei clase are propria copie a tuturor datelor membre ale clasei. În anumite cazuri, doar o copie a unei variabile trebuie partajată de toate obiectele clasei. O dată membră `static` poate fi folosită în acest scop. Ea păstrează o informație la nivelul clasei şi nu specifică unui obiect. Declarația unui membru static începe cu cuvântul cheie `static`.

Chiar dacă datele membre `static` par să fie similare variabilelor globale, ele aparțin domeniului clasei. Datele membre static pot fi `public`, `private` sau `protected`. Ele *trebuie inițializate o singură dată* în domeniul fişier. Un membru `public static` al unei clase poate fi accesat printr-un obiect al clasei sau direct prin numele clasei urmat de operatorul domeniu. Un membru static al unei clase

există chiar dacă nu există în memorie niciun obiect al clasei. Datele membre `static` pot fi accesate prin funcții membre doar dacă sunt şi ele tot `static`.

Programul de mai jos ilustrează folosirea datelor membre `private static` şi a funcțiilor membre `public static`. Data membră `count` din clasa `Employee` este declarată `static` şi este inițializată cu `0` în domeniul fişier prin instrucțiunea

```
int Employee::count = 0;
```

Data membră `count` numără câte obiecte `Employee` au fost instanțiate în program. Această valoare este incrementată de constructorul clasei şi este decrementată de destructor.

Exemplu

employee2.h

```
#ifndef EMPLOYEE2_H
#define EMPLOYEE2_H
class Employee
{
  public:
    Employee(const char*, const char*);
    ~Employee();
    const char *getFirstName() const;
    const char *getLastName() const;
    //functie membra static
    //intoarce nr. de obiecte instantiate
    static int getCount();
  private:
    char *firstName;
    char *lastName;
    //data membra static
    //numarul de obiecte instantiate
    static int count;
};
#endif
```

employee2.cpp

```
#include <iostream>
using std::cout;
using std::endl;

#include <cstring>
#include <cassert>
#include "employee2.h"

//Initializarea datei membre static
int Employee::count = 0;

int Employee::getCount() { return count; }

//Constructorul aloca dinamic spatiu pentru
//nume si prenume si foloseste strcpy pentru
//a copia numele si prenumele in obiect
Employee::Employee(const char *first, const char *last)
{
```

```
  firstName = new char[strlen(first) + 1];
  //verifica daca a fost alocata memoria
  assert(firstName != 0);
  strcpy(firstName, first);

  lastName = new char[strlen(last) + 1];
  assert(lastName != 0);
  strcpy(lastName, last);

  ++count; //incrementeaza variabila statica
  cout << "Constructorul pentru obiectul Employee "
       << firstName << ' ' << lastName << endl;
}

Employee::~Employee()
{
  cout << "~Employee() apelat pentru "
       << firstName << ' ' << lastName << endl;
  delete [] firstName;
  delete [] lastName;
  --count; //decrementeaza variabila statica
}

const char *Employee::getFirstName() const
{
  //Clientul trebuie sa copieze stringul returnat inainte
  //ca destructorul obiectului sa elibereze memoria
  return firstName;
}

const char *Employee::getLastName() const
{
  //Clientul trebuie sa copieze stringul returnat inainte
  //ca destructorul obiectului sa elibereze memoria
  return lastName;
}
```

test_static.cpp

```
#include <iostream>
using std::cout;
using std::endl;

#include "employee2.h"

int main()
{
  //se foloseste numele clasei
  cout << "Numarul de angajati inainte de instantiere este "
       << Employee::getCount() << endl;

  Employee *e1Ptr = new Employee("Susan", "Baker");
  Employee *e2Ptr = new Employee("Robert", "Jones");
```

```
   cout << "Numarul de angajati dupa instantiere este "
        << e1Ptr->getCount();

   cout << "\n\nAngajatul 1: "
        << e1Ptr->getFirstName()
        << " " << e1Ptr->getLastName()
        << "\nAngajatul 2: "
        << e2Ptr->getFirstName()
        << " " << e2Ptr->getLastName()
        << endl << endl;

   delete e1Ptr;
   e1Ptr = 0;
   delete e2Ptr;
   e2Ptr = 0;

   cout << "Numarul de angajati dupa stergere este "
        << Employee::getCount() << endl;

   return 0;
}
```

Rezulzatul rulării programului este:

```
Numarul de angajati inainte de instantiere este 0
Constructorul pentru obiectul Employee Susan Baker
Constructorul pentru obiectul Employee Robert Jones
Numarul de angajati dupa instantiere este 2

Angajatul 1: Susan Baker
Angajatul 2: Robert Jones

~Employee() apelat pentru Susan Baker
~Employee() apelat pentru Robert Jones
Numarul de angajati dupa stergere este 0
```

Atunci când încă nu a fost instanțiat niciun obiect, valoarea datei membre `count` nu poate fi referită decât prin apelul funcției membre statice `getCount`

```
Employee::getCount();
```

După instanțierea obiectului `e1Ptr`, funcția `getCount` poate fi apelată şi prin intermediul său. Apelurile `e1Ptr->getCount()` şi `Employee::getcount()` produc acelaşi rezultat.

Spre deosebire de funcțiile membre non-`static`, o funcție membră `static` nu are pointer `this` pentru ca datele membre `static` şi funcțiile membre `static` există independent de obiectele clasei.

Obiectele `e1Ptr` şi `e2Ptr` sunt alocate dinamic prin operatorul `new`. Acesta apeleaza constructorul clasei `Employee` pentru fiecare dintre ele. Prin folosirea operatorului delete se dealocă cele două obiecte şi se apelează destructorii lor.

Macro-ul `assert` definit în fişierul header `cassert` testează valoarea unei condiții. Dacă valoarea este falsă, se generează un mesaj de eroare şi programul se termină prin apelul funcției `abort` care nu mai apelează destructorii. Atunci când

contiția este adevărată, programul continuă. În exemplul de mai sus, dacă `assert` detectează faptul că operatorul new nu a putut aloca memoria, programul se încheie.

- ***Clase proxy***

Este de dorit ca detaliile de implementare să rămână ascunse clienților clasei pentru a preveni accesul la datele private şi la logica programului din clasă. Oferirea unei *clase proxy* care cunoaşte doar interfața publică a clasei permite clienților să folosească serviciile clasei fără a avea acces la detaliile de implementare.

Implementarea unei clase proxy presupune mai mulți paşi, primul fiind definirea şi implementarea clasei ale cărei date private trebuie ascunse. Clasa din exemplul de mai jos se numeşte `Implementation`, iar clasa proxy se numeşte `Interface`. Clasa `Implementation` are o dată membră privată numită `value` inițializată printr-un constructor şi două funcții numite `setValue` şi `getValue`. Clasa proxy are o interfață publică identică cu cea a clasei `Implementation` şi o singură dată membră privată numită `ptr` care este un pointer la clasa `Implementation`. Folsind acest pointer, i se ascund clientului detaliile de implementare.

Exemplu

implementation.h

```
class Implementation
{
  public:
    Implementation(int v) { value = v; }
    void setValue(int v) { value = v; }
    int getValue() const { return value; }
  private:
    int value;
};
```

interface.h

```
class Implementation;//declaratie forward

class Interface
{
  public:
    Interface(int);
    void setValue(int);  //Aceeasi interfata
    int getValue() const;//ca si Implementation
    ~Interface();
  private:
    Implementation *ptr; //necesita declaratia forward
};
```

interface.cpp

```
#include "interface.h"
#include "implementation.h"

Interface::Interface(int v)
  : ptr( new Implementation(v))
{}

void Interface::setValue(int v) {ptr->setValue(v);}
```

```
int Interface::getValue() const {return ptr->getValue();}

Interface::~Interface() {delete ptr;}
```

test_interface.cpp

```
#include <iostream>
using std::cout;
using std::endl;

#include "interface.h"

int main()
{
  Interface i(5);
  cout << "Interfata contine: " << i.getValue()
       << " inainte de setValue" << endl;

  i.setValue(10);
  cout << "Interfata contine: " << i.getValue()
       << " dupa de setValue" << endl;

  return 0;
}
```

Clasa `Interface` este clasa proxy pentru clasa `Implementation`. Singura mențiune referitoare la clasa `Implementation` este pointerul `ptr` din clasa `Interface`. Atunci când definiția unei clase foloseşte doar un pointer la o altă clasă, fişierul header care conține clasa referită nu trebuie inclus. Este nevoie doar de o declarație forward a clasei.

Deoarece fişierul `interface.cpp` este transmis clientului doar sub forma codului obiect compilat, acesta nu poate vedea interacțiunile dintre clasa proxy şi clasa proprietar.