Facultatea de Matematică şi Informatică
Universitatea din Bucureşti

Programare orientată pe obiecte

- suport de curs -

Andrei Păun
Anca Dobrovăț

An universitar 2023 – 2024
Semestrul II
Seriile 13, 14, 15

Curs 4

Cuprins

- Recapitulare curs 3
- Membrii statici ai unei clase
- Clase locale (in functii si in alte clase)
- Operatorul ::
- supraincarcarea functiilor in C++
- supraincarcarea operatorilor in C++

Variabile statice (reminder)

```
#include <iostream>

using namespace std;

void f(){
    static int x = 56; /// nu se dezaloca de fiecare data
    ///x = 56;
    x++;
    cout<<&x<<" "<<x<<'\n';
}

int main()
{
    f(); f(); f();
    return 0;
}
```

Se va afisa 57, 58, 59

Membrii statici ai unei clase

- date membre:
    - nestatice (distincte pentru fiecare obiect);
    - **statice** (unice pentru toate obiectele clasei, exista o singura copie pentru toate obiectele).

-cuvant cheie "**static**"

-create, initializate si accesate – independent de obiectele clasei.

- Variabile - alocarea si initializarea – in afara clasei.
- Const – alocarea si initializarea – in interiorul clasei

Membrii statici ai unei clase

- functiile statice:
    - efectueaza operatii asupra intregii clase;
    - nu au cuvantul cheie “this”;
    - se pot referi doar la membrii statici.

- referirea membrilor statici:
    - clasa :: membru;
    - obiect.membru (identic cu nestatic).

Folosirea uzuala a functiilor statice

```
#include <iostream>
using namespace std;
class static_type {
   static int i;
public:
   static void init(int x) { i = x;}
   void show() {cout << i;}
};
int static_type::i; // define i
int main()
{
   // init static data before object creation
   static_type::init(100);
   static_type x;
   x.show(); // displays 100
   return 0;
}
```

Obiecte statice

- Precedate de cuvantul **static**
- Se initializeaza o singura data si se utilizeaza daca se doreste pastrarea valorii la apelul multiplu al functiei
- 2 tipuri: - obiecte **locale** statice si obiecte **globale** statice
- Un obiect static local este distrus la finalul programului, dar, este vizibil doar in scope-ul in care a fost definit.
- Obiecte globale statice sunt ultimele distruse

Obiecte statice locale

Un obiect static local este distrus la finalul programului.

```
#include <iostream>

using namespace std;

class Test{ int a;
public:
    Test(int x = 0):a(x){cout<<"C "<<a<<endl;}
    ~Test(){cout<<"D "<<a<<endl;}
};

void f(int i){ static Test obiect(i); }

int main(){
    cout<<"start"<<'\n';
    f(1);
    cout<<"end"<<'\n';
}
```

```
start
C 1
end
D 1
```

Obiecte statice locale

Ce se intampla daca se apeleaza functia de mai multe ori?

```
#include <iostream>

using namespace std;

class Test{ int a;
public:
    Test(int x = 0):a(x){cout<<"C "<<a<<endl;}
    ~Test(){cout<<"D "<<a<<endl;}
};

void f(int i){ static Test obiect(i); cout<<&obiect<<endl;}

int main()
{
    cout<<"start"<<'\n';
    f(1);
    f(2);
    f(3);
    cout<<"end"<<'\n';
}
```

```
start
C 1
0x407034
0x407034
0x407034
end
D 1
```

De cate ori este creat si distrus obiectul static local? Raspuns: o singura data

Obiecte statice locale (in functii membru)

```
class Test
{
    int x;
public:
    Test(int x=0): x(x){cout<<x<<" C\n";}
    ~Test(){cout<<x<<" D\n";}
    void f()
    {
        static Test obiect;
    }
};

int main()
{
    Test obiect1(1);
    cout<<"start"<<'\n';
    obiect1.f();
    cout<<"end"<<'\n';
}
```

```
1 C
start
0 C
end
1 D
0 D
```

De cate ori este creat si distrus obiectul static local? Raspuns: o singura data

Obiecte statice globale (sunt distruse ultimele)

```
class Test
{
    int x;
public:
    Test(int x=0): x(x){cout<<x<<" C\n";}
    ~Test(){cout<<x<<" D\n";}
    void f() { static Test obiect; cout<<&obiect<<endl; }
};

static Test A(2), B(3);

int main()
{
    Test obiect1(1);
    cout<<"start"<<'\n';
    obiect1.f();
    obiect1.f();
    cout<<"end"<<'\n';
}
```

```
2 C
3 C
1 C
start
0 C
0x4030c0
0x4030c0
end
1 D
0 D
3 D
2 D
```

Operatorul de rezolutie de scop ::

```
int i; // global i

void f()

{

        int i; // local i

        i = 10; // uses local i.

}
```

```
int i; // global i
void f()
{
        int I = 7; // local i
        ::i = 10; // now refers to global i
Cout<<::I;

}
```

Clase locale

- putem defini clase in clase sau functii
- **class** este o declaratie, deci defineste un scop
- operatorul de rezolutie de scop ajuta in aceste cazuri
- rar utilizate clase in clase

```
#include <iostream>
using namespace std;
void f();
int main() {
        f(); // myclass not known here
        return 0; }
void f() {
        class myclass
        {
                int i;
        public:
                void put_i(int n) { i=n; }
                int get_i() { return i; }
        } ob;
        ob.put_i(10);
        cout << ob.get_i();
}
```

- exemplu de clasa in functia f()
- restrictii: functii definite in clasa
- nu acceseaza variabilele locale ale functiei
- acceseaza variabilele definite static
- fara variabile static definite in clasa

Mai multe detalii se regasesc in fisierul .cpp adaugat in Teams.

Functii care intorc obiecte

- o functie poate intoarce obiecte
- un obiect temporar este creat automat pentru a tine informatiile din obiectul de intors
- acesta este obiectul care este intors
- dupa ce valoarea a fost intoarsa, acest obiect este distrus
- probleme cu memoria dinamica: solutie **polimorfism pe = si pe constructorul de copiere**

```
// Returning objects from a function.
#include <iostream>
using namespace std;

class myclass
{
        int i;

public:
Myclass(){
        void set_i(int n) { i=n; }
        int get_i() { return i; }
};

myclass f(); // return object of type myclass

int main()
{
        myclass o;
        o = f();
        cout << o.get_i() << "\n";
        return 0;
}

myclass f()
{
        myclass x;
        x.set_i(1);
        return x;
}
```

copierea prin operatorul =

- este posibil sa dam valoarea unui obiect altui obiect
- trebuie sa fie de acelasi tip (aceeasi clasa)

Supraincarcarea operatorilor in C++

- majoritatea operatorilor pot fi supraincarcati
- similar ca la functii
- una din proprietatile C++ care ii confera putere
- s-a facut supraincarcarea operatorilor si pentru operatii de I/O (<<,>>)
- supraincarcarea se face definind o functie operator: membru al clasei sau nu

Restrictii

- nu se poate redefini si precedenta operatorilor
- nu se poate redefini numarul de operanzi
  - rezonabil pentru ca redefinim pentru lizibilitate
  - putem ignora un operand daca vrem
- nu putem avea valori implicite; exceptie pentru ( )
- **nu putem face overload pe . (acces de membru)**

:: (rezolutie de scop)

.*(acces membru prin pointer)

? (ternar)

sizeof()

- e bine sa facem operatiuni apropiate de intelesul operatorilor respectivi

Functii operator membri ai clasei

ret-type class-name*::operator#*(arg-list)
{
// operations
}

- # este operatorul supraincarcat (+ - * / ++ -- = , etc.)
- deobicei ret-type este tipul clasei, dar avem flexibilitate
- pentru operatori unari arg-list este vida
- pentru operatori binari: arg-list contine un element

```
class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt; }
  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }
loc operator+(loc op2);
};
```

```
// Overload + for loc.
loc loc::operator+(loc op2)
{
  loc temp;
  temp.longitude = op2.longitude + longitude;
  temp.latitude = op2.latitude + latitude;
  return temp;
}

int main(){
loc ob1(10, 20), ob2( 5, 30);
ob1.show(); // displays 10 20
ob2.show(); // displays 5 30
ob1 = ob1 + ob2;
ob1.show(); // displays 15 50
return 0;
}
```

- un singur argument pentru ca avem **this**
- longitude==this->longitude
- obiectul din stanga face apelul la functia operator
  - ob1a chemat operatorul + redefinit in clasa lui ob1

- **daca intoarcem acelasi tip de date in operator putem avea expresii**
- daca intorceam alt tip nu puteam face

  ob1 = ob1 + ob2;
- putem avea si

(ob1+ob2).show(); // displays outcome of ob1+ob2
- pentru ca functia show() este definita in clasa lui ob1
- se genereaza un obiect temporar
  - (constructor de copiere)

```
#include <iostream>
using namespace std;
class loc {   int longitude, latitude;
public:
  loc() {} // needed to construct temporaries
  loc(int lg, int lt) { longitude = lg; latitude = lt; }
  void show() { cout<<longitude<<" "<<latitude<<"\n";}
  loc operator+(loc op2);
  loc operator-(loc op2);
  loc operator=(loc op2);
  loc operator++();
};
// Overload + for loc.
loc loc::operator+(loc op2){   loc temp;
  temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
  return temp;}

loc loc::operator-(loc op2){   loc temp;
  temp.longitude = longitude - op2.longitude;
  temp.latitude = latitude - op2.latitude;
 return temp;}
```

```
// Overload asignment for loc.
loc loc::operator=(loc op2){
longitude = op2.longitude;
latitude = op2.latitude;
return *this; }// object that generated call
// Overload prefix ++ for loc.
loc loc::operator++(){
      longitude++;
      latitude++;
      return *this;}
int main(){
  loc ob1(10, 20), ob2( 5, 30), ob3(90, 90);
ob1.show();   ob2.show();
 ++ob1;   ob1.show(); // displays 11 21
 ob2 = ++ob1;   ob1.show(); // displays 12 22
ob2.show(); // displays 12 22
 ob1 = ob2 = ob3; // multiple assignment
ob1.show(); // displays 90 90
 ob2.show(); // displays 90 90
 return 0;}
```

- apelul la functia operator se face din obiectul din stanga (pentru operatori binari)
  - din aceasta cauza pentru – avem functia definita asa
- operatorul = face copiere pe variabilele de instanta, intoarce *this
- se pot face atribuiri multiple (dreapta spre stanga)

Formele prefix si postfix

- am vazut prefix, pentru postfix: definim un parametru int “dummy”

```
// Prefix increment
type operator++( ) {
    // body of prefix operator
}
```

```
// Postfix increment
type operator++( int x) {
    // body of postfix operator
}
```

Supraincarcarea +=,*=, etc.

```
loc loc::operator+=(loc op2)
    {
        longitude = op2.longitude + longitude;
        latitude = op2.latitude + latitude;
     return *this;
    }
```

- Este posibil sa facem o decuplare completa intre intelesul initial al operatorului
  - exemplu: << >>
- mostenire: operatorii (mai putin =) sunt mosteniti de clasa derivata
- clasa derivata poate sa isi redefineasca operatorii

Supraincarcarea operatorilor ca functii prieten

- operatorii pot fi definiti si ca functie nemembra a clasei
- o facem functie prietena pentru a putea accesa rapid campurile protejate
- nu avem pointerul “this”
- deci vom avea nevoie de toti operanzii ca parametri pentru functia operator
- primul parametru este operandul din stanga, al doilea parametru este operandul din dreapta

```
#include <iostream>
using namespace std;
class loc {   int longitude, latitude;
public:
  loc() {} // needed to construct temporaries
  loc(int lg, int lt) { longitude = lg; latitude = lt; }
  void show() { cout<<longitude<<" "<<latitude<<"\n";}
  friend loc operator+(loc op1, loc op2); // friend
  loc operator-(loc op2);
  loc operator=(loc op2);
  loc operator++();
};

// Now, + is overloaded using friend function.
   loc operator+(loc op1, loc op2){
      loc temp;
      temp.longitude = op1.longitude + op2.longitude;
      temp.latitude = op1.latitude + op2.latitude;
  return temp;
}
```

```
loc loc::operator-(loc op2){   loc temp;
// notice order of operands
   temp.longitude = longitude - op2.longitude;
   temp.latitude = latitude - op2.latitude;
 return temp;}

// Overload asignment for loc.
loc loc::operator=(loc op2){
longitude = op2.longitude;
latitude = op2.latitude;
return *this; }// object that generated call

loc loc::operator++(){
      longitude++;
      latitude++;
      return *this;}
int main(){
  loc ob1(10, 20), ob2( 5, 30);
  ob1 = ob1 + ob2;
  ob1.show();
return 0;}
```

Restrictii pentru operatorii definiti ca prieten

- nu se pot supraincarca = () [] sau -> cu functii prieten
- pentru ++ sau -- trebuie sa folosim referinte

Functii prieten pentru operatori unari

- pentru ++, -- folosim referinta pentru a transmite operandul
  - pentru ca trebuie sa se modifice si nu avem pointerul this
  - apel prin valoare: primim o copie a obiectului si nu putem modifica operandul (ci doar copia)

```
#include <iostream>
using namespace std;
class loc {   int longitude, latitude;
public:
  loc() {} // needed to construct temporaries
  loc(int lg, int lt) { longitude = lg; latitude = lt; }
  void show() { cout<<longitude<<" "<<latitude<<"\n";}
  loc operator=(loc op2);
  friend loc operator++(loc& op);
  friend loc operator--(loc& op);
};
// Overload assignment for loc.
loc loc::operator=(loc op2){
longitude = op2.longitude;
latitude = op2.latitude;
return *this; }// object that generated call

// Now a friend, use a reference parameter.
   loc operator++(loc& op) {
     op.longitude++;
     op.latitude++;
  return op;
}
// Make – a friend. Use reference
   loc operator--(loc& op) {
     op.longitude--;
     op.latitude--;
  return op;
}

int main(){
  loc ob1(10, 20), ob2;
  ob1.show();
   ++ob1;
   ob1.show(); // displays 11 21
   ob2 = ++ob1;
   ob2.show(); // displays 12 22
   --ob2;
   ob2.show(); // displays 11 21
return 0;}
```

pentru varianta postfix ++ --

- la fel ca la supraincarcarea operatorilor prin functii membru ale clasei: parametru int

```
// friend, postfix version of ++
friend loc operator++(loc &op, int x);
```

Diferente supraincarcarea prin membri sau prieteni

- de multe ori nu avem diferente,
  - atunci e indicat sa folosim functii membru
  - ca funcții membru: operatori unari, cei compuși (+=, *= etc)
- uneori avem insa diferente: pozitia operanzilor
  - pentru functii membru operandul din stanga apeleaza functia operator supraincarcata
  - daca vrem sa scriem expresie: 100+ob; probleme la compilare=> functii prieten
- Interesant: depinde de operator și de situație (sursa: https://stackoverflow.com/questions/4421706)

“Spaceship operator” < = > in C++20

- Se supraincarca toti operatorii de comparatie “in acelasi timp”

```
#include <compare>

class X {
  // defines ==, !=, <, >, <=, >=, <=>
  friend auto operator<=>(const X&, const X&) = default;
};
```

(sursa: https://stackoverflow.com/questions/4421706)

- in aceste cazuri trebuie sa definim doua functii de supraincarcare:
  - int + tipClasa
  - tipClasa + int

```
#include <iostream>
using namespace std;
class loc {   int longitude, latitude;
public:
   loc() {} // needed to construct temporaries
   loc(int lg, int lt) { longitude = lg; latitude = lt; }
   void show() { cout<<longitude<<" "<<latitude<<"\n";}
   loc operator=(loc op2);
   friend loc operator+(loc op1, int op2);
   friend loc operator+(int op1, loc op2);
};
// + is overloaded for loc + int.
loc operator+(loc op1, int op2){
   loc temp;
   temp.longitude = op1.longitude + op2;
   temp.latitude =op1.latitude + op2;
 return temp;}

// + is overloaded for int + loc.
loc operator+(int op1, loc op2){
   loc temp;
   temp.longitude = op1 + op2.longitude;
   temp.latitude =op1 + op2.latitude;
 return temp;}
```

```
int main(){
  loc ob1(10, 20), ob2(5, 30) , ob3(7, 14);
  ob1.show();
  ob2.show();
  ob3.show();
   ob1 = ob2 + 10; // both of these
   ob3 = 10 + ob2; // are valid
   ob1.show();
   ob3.show();

return 0;}
```

supraincarcarea new si delete

- supraincarcare op. de folosire memorie in mod dinamic pentru cazuri speciale

```
// Allocate an object.
void *operator new(size_t size){
    /* Perform allocation. Throw bad_alloc on
failure.Constructor called automatically. */
return pointer_to_memory;
}
// Delete an object.
void operator delete(void *p){
/* Free memory pointed to by p.Destructor called
automatically. */
}
```

- size_t: predefinit
- pentru new: constructorul este chemat automat
- pentru delete: destructorul este chemat automat
- supraincarcare la nivel de clasa sau globala

```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;
class loc {    int longitude, latitude;
  public:
    loc() {}
    loc(int lg, int lt)
        {longitude = lg;
    void show() { cout
cout << latitude << "\r
    void *operator ne
    void operator del
};
// new overloaded relat
void *loc::operator n
   void *p;
   cout << "In overloaded new.\n";
   p = malloc(size);
   if(!p) { bad_alloc ba; throw ba; }
 return p;}
```

```
// delete overloaded relative to loc.
void loc::operator delete(void *p){
    cout << "In overloaded delete.\n";
```

```
In overloaded new.
In overloaded new.
10 20
-10 -20
In overloaded delete.
In overloaded delete.
```

```
; }

r for p1.\n"; return 1;}
); }

r for p2.\n"; return 1;}
```

```
return 0; }
```

- daca new sau delete sunt folositi pentru alt tip de date in program, versiunile originale sunt folosite

- se poate face overload pe new si delete la nivel global
  - se declara in afara oricarei clase
  - pentru new/delete definiti si global si in clasa, cel din clasa e folosit pentru elemente de tipul clasei, si in rest e folosit cel redefinit global

```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;
class loc { int longitude, latitude;
  public:
    loc() {}
    loc(int lg, int lt)
        {longitude = lg;latitude = lt;}
    void show() {cout << longitude << " ";
        cout << latitude << "\n";}
};
// Global new
void *operator new(size_t size) {
  void *p;
  p = malloc(size);
  if(!p) { bad_alloc ba; throw ba; }
return p;
}
// Global delete
void operator delete(void *p) { free(p); }
int main(){
  loc *p1, *p2;
  float *f;
  try {p1 = new loc (10, 20); }
  catch (bad_alloc xa) {
    cout << "Allocation error for p1.\n“;
    return 1; }
  try {p2 = new loc (-10, -20); }
  catch (bad_alloc xa) {
    cout << "Allocation error for p2.\n“;
    return 1; }
  try {
    f = new float; // uses overloaded new, too }
  catch (bad_alloc xa) {
    cout << "Allocation error for f.\n";
    return 1; }
  *f = 10.10F;
   cout << *f << "\n";
   p1->show();
   p2->show();
   delete p1; delete p2; delete f;
return 0; }
```

new si delete pentru array-uri

- facem overload de doua ori

```
// Allocate an array of objects.
void *operator new[](size_t size) {
   /* Perform allocation. Throw bad_alloc on failure.
Constructor for each element called automatically. */
return pointer_to_memory;
}
 // Delete an array of objects.
void operator delete[](void *p) {
/* Free memory pointed to by p. Destructor for each
element called automatically. */
}
```

supraincarcarea []

- trebuie sa fie functii membru, (nestatice)
- nu pot fi functii prieten
- este considerat operator binar
- o[3] se tranfsorma in
- o.operator[](3)

```
type class-name::operator[](int i)
{
// . . .
}
```

supraincarcarea []

```
#include <iostream>
using namespace std;
class atype { int a[3];
public:
  atype(int i, int j, int k) { a[0] = i; a[1] = j; a[2] = k; }
  int operator[](int i) { return a[i]; }
};
int main() {
  atype ob(1, 2, 3);
  cout << ob[1]; // displays 2
return 0;
}
```

- operatorul [] poate fi folosit si la stanga unei atribuiri (obiectul intors este atunci referinta)

supraincarcarea []

```
#include <iostream>
using namespace std;
class atype { int a[3];
public:
  atype(int i, int j, int k) { a[0] = i; a[1] = j; a[2] = k; }
  int &operator[](int i) { return a[i]; }
};
int main() {
  atype ob(1, 2, 3);
  cout << ob[1]; // displays 2
  cout << " ";
  ob[1] = 25; // [] on left of =
  cout << ob[1]; // now displays 25
return 0; }
```

- putem in acest fel verifica array-urile
- exemplul urmator

```
// A safe array example.
#include <iostream>
#include <cstdlib>
using namespace std;
class atype { int a[3];
public:
  atype(int i, int j, int k) {a[0] = i; a[1] = j; a[2] = k;}
 int &operator[](int i);
};

// Provide range checking for atype.
int &atype::operator[](int i)
{
   if(i<0 || i> 2) { cout << "Boundary Error\n"; exit(1); }
return a[i];
}
```

```
int main() {
   atype ob(1, 2, 3);
   cout << ob[1]; // displays 2
   cout << " ";
   ob[1] = 25; // [] appears on left
   cout << ob[1]; // displays 25
   ob[3] = 44;
            // generates runtime error, 3 out-of-range
  return 0; }
```

supraincarcarea ()

- nu creem un nou fel de a chema functii
- definim un mod de a chema functii cu numar arbitrar de parametrii

```
double operator()(int a, float f, char *s);
O(10, 23.34, "hi");
                    echivalent cu O.operator()(10, 23.34, "hi");
```

supraincarcarea ()

```
#include <iostream>
using namespace std;
class loc { int longitude, latitude;
 public:
    loc() {}
    loc(int lg, int lt) {longitude = lg;
latitude = lt;}
    void show() {cout << longitude << " ";
cout << latitude << "\n";}
    loc operator+(loc op2);
    loc operator()(int i, int j);
 };
// Overload ( ) for loc.
loc loc::operator()(int i, int j) {
   longitude = i; latitude = j;
return *this;
}
```

```
 Overload + for loc.
)c loc::operator+(loc op2) {
 loc temp;
 temp.longitude = op2.longitude + longitude;
 temp.latitude = op2.latitude + latitude; return
:mp;

nt main() { loc ob1(10, 20), ob2(1, 1);
b1.show();
b1(7, 8); // can be executed by itself ob1.show();
b1 = ob2 + ob1(10, 10); // can be used in
xpressions
b1.show();
eturn 0; }
```

10 20
7 8
11 11

overload pe ->

- operator unar
- obiect->element
  - obiect genereaza apelul
  - element trebuie sa fie accesibil
  - intoarce un pointer catre un obiect din clasa

```
#include <iostream>
using namespace std;
class myclass {
    public:
    int i;
    myclass *operator->() {return this;}
};

int main() {
   myclass ob; ob->i = 10; // same as ob.i
   cout << ob.i << " " << ob->i;
return 0;
}
```

supraincarcarea operatorului ,

- operator binar
- ar trebui ignorate toate valorile mai putin a celui mai din dreapta operand

```
#include <iostream>
using namespace std;
class loc {   int longitude, latitude;
  public:
    loc() {}
    loc(int lg, int lt) {longitude = lg; latitude = lt;
    void show() {cout << longitude << " ";
cout << latitude << "\n";}
    loc operator+(loc op2);
    loc operator,(loc op2);
};
// overload comma for loc
  loc loc::operator,(loc op2){
      loc temp;
      temp.longitude = op2.longitude;
      temp.latitude = op2.latitude;
    cout << op2.longitude << " “;
    cout << op2.latitude << "\n";
return temp;
}
```

```
// Overload + for loc
loc loc::operator+(loc op2) {
   loc temp;
   temp.longitude = op2.longitude + longitude;
   temp.latitude = op2.latitude + latitude;
 return temp; }

int main() {
loc ob1(10, 20), ob2( 5, 30), ob3(1, 1); ob1.show();
ob2.show(); ob3.show();
cout << "\n";
ob1 = (ob1, ob2+ob2, ob3);
ob1.show(); // displays 1 1, the value of ob3
return 0;
}
```

```
10 20
5 30
1 1
10 60
1 1
1 1
```

Perspective

Curs 5

Proiectarea descendenta a claselor. Moştenirea în C++.

1 Controlul accesului la clasa de bază.

2 Constructori, destructori şi moştenire.

3 Redefinirea membrilor unei clase de bază într-o clasă derivată.

4 Declaraţii de acces.