Facultatea de Matematică şi Informatică
Universitatea din Bucureşti

Programare orientată pe obiecte

- suport de curs -

Andrei Păun

Anca Dobrovăț

An universitar 2019 – 2020
Semestrul II

Seriile 13, 14 şi 21

Curs 4

10-11/3/2020

Cuprins

- Static, clase locale
- Constructori-destructori, CC
- Operatorul ::

- supraincarcarea functiilor in C++
- supraincarcarea operatorilor in C++

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.

- alocarea si initializarea – in afara 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;
}
```

Chestiuni despre constructori si destructori

- constructor: executat la crearea obiectului
- destructor : executat la distrugerea obiectului

- obiecte globale: constructorii executati in ordinea in care sunt definite obiectele
  - destructorii: dupa ce main s-a terminat in ordinea inversa a constructorilor

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

class myclass {   int who;
public:
            myclass(int id);
            ~myclass();
} glob_ob1(1), glob_ob2(2);

myclass::myclass(int id) {
            cout << "Initializing " << id << "\n";
            who = id; }
myclass::~myclass() { cout << "Destructing " << who << "\n";}

int main() {
            myclass local_ob1(3);
            cout << "This will not be first line displayed.\n";
            myclass local_ob2(4);
return 0;
}
```

```
Initializing 1
Initializing 2
Initializing 3
This will not be
first line displayed.
Initializing 4
Destructing 4
Destructing 3
Destructing 2
Destructing 1
```

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; // local i

        ::i = 10; // now refers to global 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

transmitere de obiecte catre functii

- similar cu tipurile predefinite
- call-by-value
- constructori-destructori!

Facultatea de Matematică şi Informatică
Universitatea din Bucureşti

```
// Passing an object to a function.
#include <iostream>
using namespace std;
class myclass {
        int i;
public:
        myclass(int n);
        ~myclass();
        void set_i(int n) { i=n; }
        int get_i() { return i; }
};

myclass::myclass(int n)
{i = n; cout << "Constructing " << i << "\n“; }
myclass::~myclass()
{cout << "Destroying " << i << "\n“; }
void f(myclass ob);

int main() {
        myclass o(1);
        f(o);
        cout << "This is i in main: ";
        cout << o.get_i() << "\n";
        return 0;
}

void f(myclass ob)
{
    ob.set_i(2);
    cout << "This is local i: " << ob.get_i();
    cout << "\n";
}
```

```
Constructing 1
This is local i: 2
Destroying 2
This is i in main: 1
Destroying 1
```

Discutie

- apelam constructorul cand cream obiectul o
- apelam de DOUA ori destructorul
- la apel de functie: apel prin valoare, o copie a obiectului e creata
  - apelam constructorul?
  - la finalizare apelam destructorul?

- La apel de functie constructorul “normal” NU ESTE APELAT

- se apeleaza asa-numitul “constructorul de copiere”
- un asemenea constructor defineste cum se copiaza un obiect

- se poate defini explicit de catre programator
  - daca nu e definit C++ il creeaza automat

Constructor de copiere

- C++ il defineste pentru a face o copie identica pe date
- constructorul e folosit pentru initializare
- constr. de copiere e folosit pentru obiect deja initializat, doar copiaza
- vrem sa folosim starea curenta a obiectului, nu starea initiala a unui obiect din clasa respectiva

Destructori pentru obiecte transmise catre functii

- trebuie sa distrugem obiectul respectiv
- chemam destructorul

- putem avea probleme cu obiecte care **folosesc memoria dinamic**: la distrugere copia elibereaza memoria, obiectul din main este defect (nu mai are memorie alocata)
- in aceste cazuri trebuie sa redefinim operatorul de copiere (copy constructor)

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:
        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 functiilor

- este folosirea aceluiasi nume pentru functii diferite
- functii diferite, dar cu inteles apropiat
- compilatorul foloseste numarul si tipul parametrilor pentru a diferentia apelurile

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

int myfunc(int i); // these differ in types of parameters
double myfunc(double i);

int main() {
        cout << myfunc(10) << " "; // calls myfunc(int i)
        cout << myfunc(5.4); // calls myfunc(double i)
        return 0;
}

double myfunc(double i) { return i; }

int myfunc(int i) { return i; }
```

tipuri diferite pentru parametrul i

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

int myfunc(int i); // these differ in number of parameters
int myfunc(int i, int j);

int main()
{
        cout << myfunc(10) << " "; // calls myfunc(int i)
        cout << myfunc(4, 5); // calls myfunc(int i, int j)
        return 0;
}

int myfunc(int i) {return i;}

int myfunc(int i, int j) {return i*j;}
```

numar diferit de parametri

- daca diferenta este doar in tipul de date intors: eroare la compilare

```
int myfunc(int i); // Error: differing return types are
float myfunc(int i); // insufficient when overloading.
```

- sau tipuri care _par_ sa fie diferite

```
void f(int *p);
void f(int p[]); // error, *p is same as p[]

void f(int x);
void f(int& x);
```

Polimorfism pe constructori

- foarte comun sa fie supraincarcati
- de ce?
  - flexibilitate
  - pentru a putea defini obiecte initializate si neinitializate
  - constructori de copiere: copy constructors

overload pe constructori: flexibilitate

- putem avea mai multe posibilitati pentru initializarea/construirea unui obiect
- definim constructori pentru toate modurile de initializare
- daca se incearca initializarea intr-un alt fel (decat cele definite): eroare la compilare

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

class date { int day, month, year;
public:
        date(char *d);
        date(int m, int d, int y);
        void show_date();
};

// Initialize using string.
date::date(char *d)
{ sscanf(d, "%d%*c%d%*c%d", &month, &day, &year);}

// Initialize using integers.
date::date(int m, int d, int y)
{ day = d; month = m; year = y; }
```

```
void date::show_date()
{
     cout << month << "/" << day;
     cout << "/" << year << "\n";
}

int main()
{
  date ob1(12, 4, 2003), ob2("10/22/2003");
  ob1.show_date();
  ob2.show_date();
  return 0;
}
```

```
int main()
{
  char s[80];
  cout << "Enter new date: ";
  cin >> s;
  date d(s);
  d.show_date();
  return 0;
}
```

"%d%*c%d%*c%d

citim din sir
*: ignoram ce citim
c: un singur caracter

citim 3 intregi sau
luna/zi/an

polimorfism de constructori: obiecte initializate si ne-initializate

- important pentru array-uri dinamice de obiecte
- nu se pot initializa obiectele dintr-o lista alocata dinamic
- asadar avem nevoie de posibilitatea de a crea obiecte neinitializate (din lista dinamica) si obiecte initializate (definite normal)

```
#include <iostream>
#include <new>
using namespace std;
class powers
{  int x;
public:
        // overload constructor two ways
};

int main()
{
  powers ofTwo[] = {1, 2, 4, 8, 16}; // initialized
  powers ofThree[5]; // uninitialized
  powers *p;
  int i; // show powers of two
  cout << "Powers of two: ";
  for(i=0; i<5; i++) {
        cout << ofTwo[i].getx() << " ";
  }
cout << "\n\n";
// set powers of three
ofThree[0].setx(1); ofThree[1].setx(3);
ofThree[2].setx(9); ofThree[3].setx(27);
ofThree[4].setx(81);

// show powers of three
cout << "Powers of three: ";
for(i=0; i<5; i++) { cout << ofThree[i].getx() << " ";}
cout << "\n\n";
```

- ofThree si lista p au nevoie de constructorul fara parametri

```
return 1;}

// initialize dynamic array with powers of two
for(i=0; i<5; i++) { p[i].setx(ofTwo[i].getx());}

// show powers of two
cout << "Powers of two: ";
for(i=0; i<5; i++) {cout << p[i].getx() << " ";}
cout << "\n\n";
delete [] p;
return 0;
}
```

polimorfism de constructori: constructorul de copiere

- pot aparea probleme cand un obiect initializeaza un alt obiect

```
MyClass B = A;
```

- aici se copiaza toate campurile (starea) obiectului A in obiectul B
- problema apare la alocare dinamica de memorie: A si B folosesc aceeasi zona de memorie pentru ca pointerii arata in acelasi loc
- destructorul lui MyClass elibereaza aceeasi zona de memorie de doua ori (distruge A si B)

constructorul de copiere

- aceeasi problema
  - apel de functie cu obiect ca parametru
  - apel de functie cu obiect ca variabila de intoarcere
    - in aceste cazuri un obiect temporar este creat, se copiaza prin constructorul de copiere in obiectul temporar, si apoi se continua
    - deci vor fi din nou doua distrugeri de obiecte din clasa respectiva (una pentru parametru, una pentru obiectul temporar)

putem redefini constructorul de copiere

```
classname (const classname &o) {
// body of constructor
}
```

- o este obiectul din dreapta
- putem avea mai multi parametri (dar trebuie sa definim valori implicite pentru ei)
- & este apel prin referinta
- putem avea si atribuire (o1=o2;)
  - redefinim operatorii mai tarziu, putem redefini =
  - = diferit de initializare

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

class array {
  int *p;
  int size;
public:
  array(int sz) {
              try {
                p = new int[sz];
              } catch (bad_alloc xa) {
                cout << "Allocation Failure\n";
                exit(EXIT_FAILURE);
              }
              size = sz;
  }
  ~array() { delete [] p; }

  // copy constructor
  array(const array &a);
  void put(int i, int j) {if(i>=0 && i<size) p[i] = j;}
  int get(int i) { return p[i];}
};
```

```
// Copy Constructor
array::array(const array &a) {
              int i;
              try {
                p = new int[a.size];
              } catch (bad_alloc xa) {
                cout << "Allocation Failure\n";
                exit(EXIT_FAILURE);
              }
              for(i=0; i<a.size; i++) p[i] = a.p[i];
}
```

```
int main()
{
              array num(10);
              int i;
              for(i=0; i<10; i++) num.put(i, i);
              for(i=9; i>=0; i--) cout << num.get(i);
              cout << "\n";
// create another array and initialize with num
              array x(num); // invokes copy constructor
               for(i=0; i<10; i++) cout << x.get(i);
               return 0;
}
```

- Observatie: constructorul de copiere este folosit doar la initializari

- daca avem

```
array a(10);
array b(10);
b=a;
```

  - nu este initializare, este copiere de stare
  - este posibil sa trebuiasca redefinit si operatorul = (mai tarziu)

pointeri catre functii polimorfice

- putem avea pointeri catre functii (C)
- putem avea pointeri catre functii polimorfice

- cum se defineste pointerul ne spune catre ce versiune a functiei cu acelasi nume aratam

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

int myfunc(int a);
int myfunc(int a, int b);

int main()
{
        int (*fp)(int a); // pointer to int f(int)
        fp = myfunc; // points to myfunc(int)
        cout << fp(5);
        return 0;
}

int myfunc(int a)
{
        return a;
}

int myfunc(int a, int b)
{
        return a*b;
}
```

- semnatura functiei din definitia pointerului ne spune ca mergem spre functia cu un parametru
  - trebuie sa existe una din variantele polimorfice care este la fel cu definitia pointerului

Argumente implicite pentru functii

- putem defini valori implicite pentru parametrii unei functii
- valorile implicite sunt folosite atunci cand acei parametri nu sunt dati la apel

```
void myfunc(double d = 0.0)
{
 // ...
 }
```

- …

```
myfunc(198.234); // pass an explicit value
myfunc(); // let function use default
```

Argumente implicite

- dau posibilitatea pentru flexibilitate
- majoritatea functiilor considera cel mai general caz, cu parametrii impliciti putem sa chemam o functie pentru cazuri particulare
- multe functii de I/O folosesc arg. implicite
- nu avem nevoie de overload

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

void clrscr(int size=25);

int main()
{
        register int i;
        for(i=0; i<30; i++ ) cout << i << endl;
        cin.get();
        clrscr(); // clears 25 lines
        for(i=0; i<30; i++ ) cout << i << endl;
        cin.get();
        clrscr(10); // clears 10 lines
        return 0;
}

void clrscr(int size)
{
        for(; size; size--) cout << endl;
}
```

- se pot refolosi valorile unor parametri

```
void iputs(char *str, int indent)
{
        if(indent < 0) indent = 0;
        for( ; indent; indent--) cout << " ";
        cout << str << "\n";
}
```

```
#include <iostream>
using namespace std;
/* Default indent to -1. This value tells the function to reuse the previous value. */
void iputs(char *str, int indent = -1);

int main() {
        iputs("Hello there", 10);
        iputs("This will be indented 10 spaces by default");
        iputs("This will be indented 5 spaces", 5);
        iputs("This is not indented", 0);
        return 0;}

void iputs(char *str, int indent)
{
        static i = 0; // holds previous indent value
        if(indent >= 0)
                i = indent;
        else // reuse old indent value
                indent = i;
        for( ; indent; indent--) cout << " ";
        cout << str << "\n";
}
```

Hello there
This will be indented 10 spaces by default
This will be indented 5 spaces
This is not indented

parametri impliciti

- se specifica o singura data
- pot fi mai multi
- toti sunt la dreapta

- putem avea param. impliciti in definitia constructorilor
  - nu mai facem overload pe constructor
  - nu trebuie sa ii precizam mereu la declarare

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

class cube {
        int x, y, z;
public:
        cube(int i=0, int j=0, int k=0) {
                x=i;
                y=j;
                z=k;
        }

        int volume() {
                return x*y*z;
        }
};

int main()
{
        cube a(2,3,4), b;
        cout << a.volume() << endl;
        cout << b.volume();
        return 0;
}
```

cube() {x=0; y=0; z=0}

```
// A customized version of strcat().
#include <iostream>
#include <cstring>
using namespace std;

void mystrcat(char *s1, char *s2, int len = -1);

int main()
{
  char str1[80] = "This is a test";
  char str2[80] = "0123456789";
  mystrcat(str1, str2, 5); // concatenate 5 chars
  cout << str1 << '\n';
  strcpy(str1, "This is a test"); // reset str1
  mystrcat(str1, str2); // concatenate entire string
  cout << str1 << '\n';

  return 0;
}
```

```
// A custom version of strcat().
void mystrcat(char *s1, char *s2, int len)
{   // find end of s1
 while(*s1) s1++;
 if(len == -1) len = strlen(s2);
while(*s2 && len)
{
      *s1 = *s2; // copy chars
       s1++;
       s2++;
       len--;
}
*s1 = '\0'; // null terminate s1
}
```

parametri impliciti

- modul corect de folosire este de a defini un asemenea parametru cand se subintelege valoarea implicita
- daca sunt mai multe posibilitati pentru valoarea implicita e mai bine sa nu se foloseasca (lizibilitate)
- cand se foloseste un param. implicit nu trebuie sa faca probleme in program

Ambiguitati pentru polimorfism de functii

- erori la compilare
- majoritatea datorita conversiilor implicite

```
int myfunc(double d);
// ...
cout << myfunc('c'); // not an error, conversion applied
```

```
#include <iostream>
using namespace std;
float myfunc(float i);
double myfunc(double i);

int main(){
cout << myfunc(10.1) << " "; // unambiguous, calls myfunc(double)
cout << myfunc(10); // ambiguous
 return 0;
}

float myfunc(float i){   return i;}
double myfunc(double i){   return -i;}
```

- problema nu e de definire a functiilor myfunc,
- problema apare la apelul functiilor

```
#include <iostream>
using namespace std;
char myfunc(unsigned char ch);
char myfunc(char ch);

int main(){
cout << myfunc('c'); // this calls myfunc(char)
cout << myfunc(88) << " "; // ambiguous
return 0;
}
char myfunc(unsigned char ch)
{
    return ch-1;
}
char myfunc(char ch){return ch+1;}
```

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

int myfunc(int i);
int myfunc(int i, int j=1);

 int main()
{
 cout << myfunc(4, 5) << " "; // unambiguous
 cout << myfunc(10); // ambiguous
 return 0;
}

int myfunc(int i)
{
  return i;
}

int myfunc(int i, int j)
{
  return i*j;
}
```

- **ambiguitate intre char si unsigned char**
- **ambiguitate pentru functii cu param. impliciti**

```
// This program contains an error. #include
<iostream>
using namespace std;
void f(int x);
void f(int &x); // error

int main() {
int a=10;
f(a); // error, which f()?
return 0;
}

void f(int x) { cout << "In f(int)\n"; }
void f(int &x) { cout << "In f(int &)\n"; }
```

- doua tipuri de apel: prin valoare si prin referinta, ambiguitate!
- mereu eroare de ambiguitate

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

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;
   }
```

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)

- e bine sa facem operatiuni apropiate de intelesul operatorilor respectivi

- 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
- 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

- 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 rela
void *loc::operator n
   void *p;
   cout << "In overloa
   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";
    free(p);
```

- 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;}
```

```
 delete p1; delete p2;
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)
{
// . . .
}
```

```
#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)

```
#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");

```
#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.
oc loc::operator+(loc op2) {
 loc temp;
 temp.longitude = op2.longitude + longitude;
 temp.latitude = op2.latitude + latitude; return
emp;


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();
            r(); ob3.show();
            '\n";
            )1, ob2+ob2, ob3);
            r(); // displays 1 1, the value of ob3
            ;
```

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