



Programare orientata pe obiecte

- suport de curs -

Dobrovat Anca - Madalina

An universitar 2019 – 2020

Semestrul II

Seria 13

Curs 5

18/03/2020



Agenda cursului

Supraîncărcarea operatorilor.

- Supraîncărcarea operatorilor cu funcții friend.
- Supraîncărcarea operatorilor cu funcții membru.

Conversia datelor în C++.

- Conversii între diferite tipuri de obiecte (operatorul cast, operatorul= și constructor de copiere).
- Membrii constanți ai unei clase în C++.
- Modificatorul const, obiecte constante, pointeri constanți la obiecte și pointeri la obiecte constante.

Exemplele sunt luate din cartea **“Thinking in C++” – Bruce Eckel**



Supraincercarea operatorilor

- una din cele mai importante caracteristici ale C++;
 - majoritatea operatorilor pot fi supraincarcati, similar ca la functii;
 - NU se pot crea operatori noi
- NU SE POATE MODIFICA PRIN SUPRAINCARCARE:**
- pluralitatea operatorului (numarul de argumente);
 - precedenta si asociativitatea operatorului.

Obs: NU SE POT SUPRAINCARCA

- “ . ” (acces la membru);
 - “ .*” (acces la membru prin pointer);
 - “ :: ” (rezolutie de scop);
 - “ ? : ” (operatorul ternar).
- se face definind ***o functie operator;***
- 2 modalitati: - ca functie membra a clasei sau ca functie prietena a clasei.



Supraincercarea operatorilor

Crearea unei functii membre:

- ***sintaxa generala:***

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

Unde:

: operatorul supraincarcat (+ - * / ++ -- = , etc.);

ret-type, in general, este tipul clasei;

pentru operatori unari arg-list este vida, pentru binari arg-list contine un element.



Supraincercarea operatorilor

Ca functii membre

Obs. Operatorii:

“ = “ (atribuire);

“ [] “ (indexare);

“ () “ (apel de functie);

“ → “ (acces membru de tip pointer);

Pot fi definiti DOAR CU FUNCTII MEMBRE NESTATICE.



Supraincercarea operatorilor

Ca functii membre

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

class Integer {
    int i;
public:
    Integer(int ii) : i(ii) {}
    const Integer operator+(const Integer& rv) const {
        cout << "operator+" << endl;
        return Integer(i + rv.i);
    }
    Integer& operator+=(const Integer& rv) {
        cout << "operator+=" << endl;
        i += rv.i;
        return *this;
    }
};

int main() {
    cout << "built-in types:" << endl;
    int i = 1, j = 2, k = 3;
    k += i + j;
    cout << "user-defined types:" << endl;
    Integer ii(1), jj(2), kk(3);
    kk += ii + jj;
}
```

Operatorii += si + definiti ca functii membre inline.

Operatorii binari au un singur parametru
Operatorii unari nu au argumente.

Operatorii conditionali returneaza, de regula, o valoare booleana.

Pentru operatorii neconditionali se returneaza, de regula, un obiect sau o referinta de acelasi tip cu parametrii din lista de argumente (daca au acelasi tip)



Supraincercarea operatorilor

Ca functii membre

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

class Integer {
    int i;
public:
    Integer(int ii) : i(ii) {}
    const Integer operator+(const Integer& rv) const {
        cout << "operator+" << endl;
        return Integer(i + rv.i);
    }
    Integer& operator+=(const Integer& rv) {
        cout << "operator+=" << endl;
        i += rv.i;
        return *this;
    }
};

int main() {
    cout << "built-in types:" << endl;
    int i = 1, j = 2, k = 3;
    k += i + j;
    cout << "user-defined types:" << endl;
    Integer ii(1), jj(2), kk(3);
    kk += ii + jj;
}
```

kk = ii + jj;

Operatorul+ returneaza un **Integer** (temporar) care este utilizat ca argument pentru operatorul +=.

Obiectul temporar este distrus cand nu mai e nevoie de el.



Supraincercarea operatorilor unari

Ca functii membre (implicit "this")

```
class Byte {  
    unsigned char b;  
public:  
    Byte(unsigned char bb = 0) : b(bb) {}  
    // No side effects: const member function:  
    const Byte& operator+() const {  
        cout << "+Byte\n";  
        return *this;  
    }  
    const Byte operator-() const {  
        cout << "-Byte\n";  
        return Byte(-b);  
    }  
  
    Byte operator!() const {  
        cout << "!Byte\n";  
        return Byte(!b);  
    }  
  
    Byte* operator&() {  
        cout << "&Byte\n";  
        return this;  
    }  
};
```

```
// Side effects: non-const member function:  
const Byte& operator++() { // Prefix  
    cout << "++Byte\n";  
    b++;  
    return *this;  
}  
const Byte operator++(int) { // Postfix  
    cout << "Byte++\n";  
    Byte before(b);  
    b++;  
    return before;  
}  
const Byte& operator--() { // Prefix  
    cout << "--Byte\n";  
    --b;  
    return *this;  
}  
const Byte operator--(int) { // Postfix  
    cout << "Byte--\n";  
    Byte before(b);  
    --b;  
    return before;  
}  
};
```




Supraincarcarea operatorilor unari

Ca functii membre (implicit "this")

```
const Byte& operator++() { // Prefix
    cout << "++Byte\n";
    b++;
    return *this;
}
```

Parametru "dummy" pentru a diferentia postfix de prefix

```
const Byte operator++(int) { // Postfix
    cout << "Byte++\n";
    Byte before(b);
    b++;
    return before;
}
```

-obiect temporar creat prin constructorul parametrizat

-Se incrementeaza si se returneaza rezultatul.



Supraincercarea operatorilor unari

Ca functii friend (globale)

```
class Integer {  
    long i;  
    Integer* This() { return this; }  
public:  
    Integer(long ll = 0) : i(ll) {}  
    // No side effects takes const& argument:  
    friend const Integer& operator+(const Integer& a);  
    friend const Integer operator-(const Integer& a);  
    friend const Integer operator~(const Integer& a);  
    friend Integer* operator&(Integer& a);  
    friend int operator!(const Integer& a);  
    // Side effects have non-const& argument:  
    // Prefix:  
    friend const Integer& operator++(Integer& a);  
    // Postfix:  
    friend const Integer operator++(Integer& a, int);  
    // Prefix:  
    friend const Integer& operator--(Integer& a);  
    // Postfix:  
    friend const Integer operator--(Integer& a, int);  
};
```

```
const Integer& operator+(const Integer& a) {  
    cout << "+Integer\n";  
    return a; // Unary + has no effect  
}  
const Integer operator-(const Integer& a) {  
    cout << "-Integer\n";  
    return Integer(-a.i);  
}  
const Integer operator~(const Integer& a) {  
    cout << "~Integer\n";  
    return Integer(~a.i);  
}  
Integer* operator&(Integer& a) {  
    cout << "&Integer\n";  
    return a.This(); // &a is recursive!  
}  
int operator!(const Integer& a) {  
    cout << "!Integer\n";  
    return !a.i;  
}
```



Supraincercarea operatorilor unari

Ca functii friend (globale)

```
class Integer {  
    long i;  
    Integer* This() { return this; }  
public:  
    Integer(long ll = 0) : i(ll) {}  
    // No side effects takes const& argument:  
    friend const Integer& operator+(const Integer& a);  
    friend const Integer operator-(const Integer& a);  
    friend const Integer operator~(const Integer& a);  
    friend Integer* operator&(Integer& a);  
    friend int operator!(const Integer& a);  
    // Side effects have non-const& argument:  
    // Prefix:  
    friend const Integer& operator++(Integer& a);  
    // Postfix:  
    friend const Integer operator++(Integer& a, int);  
    // Prefix:  
    friend const Integer& operator--(Integer& a);  
    // Postfix:  
    friend const Integer operator--(Integer& a, int);  
};
```

```
// Prefix; return incremented value  
const Integer& operator++(Integer& a) {  
    cout << "++Integer\n";  
    a.i++;  
    return a;  
}  
  
// Postfix; return the value before increment:  
const Integer operator++(Integer& a, int) {  
    cout << "Integer++\n";  
    Integer before(a.i);  
    a.i++;  
    return before;  
}  
  
// Prefix; return decremented value  
const Integer& operator--(Integer& a) {  
    cout << "--Integer\n";  
    a.i--;  
    return a;  
}  
  
// Postfix; return the value before decrement:  
const Integer operator--(Integer& a, int) {  
    cout << "Integer--\n";  
    Integer before(a.i);  
    a.i--;  
    return before;  
}
```



Supraincercarea operatorilor unari

Ca functii membre (implicit “this”)

```
const Integer& operator++(Integer& a) {  
    cout << "++Integer\n";  
    a.i++;  
    return a;  
}
```

Parametru “dummy” pentru a diferentia postfix de prefix

```
const Integer operator++(Integer& a, int) {  
    cout << "Integer++\n";  
    Integer before(a.i);  
    a.i++;  
    return before;  
}
```

-obiect temporar creat prin constructorul parametrizat

-Se incrementeaza si se returneaza rezultatul.



Supraincercarea operatorilor binari

Ca functii membre (implicit "this")

```
class Byte {  
    unsigned char b;  
public:  
    Byte(unsigned char bb = 0) : b(bb) {}  
    // No side effects: const member function:  
    const Byte operator+(const Byte& right) const { return Byte(b + right.b); }  
    const Byte operator-(const Byte& right) const { return Byte(b - right.b); }  
    // operatori pe biti  
    const Byte operator&(const Byte& right) const { return Byte(b & right.b); }  
    const Byte operator|(const Byte& right) const { return Byte(b | right.b); }  
    const Byte operator<<(const Byte& right) const { return Byte(b << right.b); }
```



Supraincarcarea operatorilor binari

Ca functii membre (implicit “this”)

```
// Assignments modify & return lvalue.  
// operator= can only be a member function:  
Byte& operator=(const Byte& right) { // Handle self-assignment:  
    if(this == &right) return *this;  
    b = right.b;  
    return *this;  
}  
Byte& operator+=(const Byte& right) {  
    if(this == &right) { /* self-assignment */  
        b += right.b;  
        return *this;  
    }  
    Byte& operator-=(const Byte& right) {  
        if(this == &right) { /* self-assignment */  
            b -= right.b;  
            return *this;  
        }  
    }
```



Supraincercarea operatorilor binari

Ca functii membre (implicit “this”)

```
// Conditional operators return true/false:
int operator==(const Byte& right) const {
    return b == right.b;
}
int operator!=(const Byte& right) const {
    return b != right.b;
}
int operator<(const Byte& right) const {
    return b < right.b;
}
int operator>(const Byte& right) const {
    return b > right.b;
}
int operator<=(const Byte& right) const {
    return b <= right.b;
}
int operator>=(const Byte& right) const {
    return b >= right.b;
}
int operator&&(const Byte& right) const {
    return b && right.b;
}
int operator||(const Byte& right) const {
    return b || right.b;
}
```



Supraincercarea operatorilor binari

Ca functii friend

```
class Integer {  
    long i;  
public:  
    Integer(long ll = 0) : i(ll) {}  
    // Operators that create new, modified value:  
    friend const Integer operator+(const Integer& left, const Integer& right);  
    friend const Integer operator-(const Integer& left, const Integer& right);  
  
    friend const Integer operator&(const Integer& left, const Integer& right);  
    friend const Integer operator|(const Integer& left, const Integer& right);  
  
    // Assignments modify & return lvalue:  
    friend Integer& operator+=(Integer& left, const Integer& right);  
    friend Integer& operator-=(Integer& left, const Integer& right);  
  
    // Conditional operators return true/false:  
    friend int operator==(const Integer& left, const Integer& right);  
    friend int operator!=(const Integer& left, const Integer& right);  
    friend int operator<(const Integer& left, const Integer& right);  
  
};
```




Supraincercarea operatorilor binari

Ca functii friend

```
const Integer operator+(const Integer& left, const Integer& right) { return Integer(left.i + right.i); }
const Integer operator-(const Integer& left, const Integer& right) { return Integer(left.i - right.i); }
const Integer operator&(const Integer& left, const Integer& right) { return Integer(left.i & right.i); }
const Integer operator~(const Integer& left, const Integer& right) { return Integer(left.i ~ right.i); }
```

// Assignments modify & return lvalue:

```
Integer& operator+=(Integer& left, const Integer& right) {
    if(&left == &right) { /* self-assignment */
        left.i += right.i;
    }
    return left;
}
```

```
Integer& operator-=(Integer& left, const Integer& right) {
    if(&left == &right) { /* self-assignment */
        left.i -= right.i;
    }
    return left;
}
```

// Conditional operators return true/false:

```
int operator==(const Integer& left, const Integer& right) { return left.i == right.i; }
int operator!=(const Integer& left, const Integer& right) { return left.i != right.i; }
int operator<(const Integer& left, const Integer& right) { return left.i < right.i; }
```



Supraincercarea operatorilor

Argumente si valori returnate

1. Daca NU se doreste modificarea unui argument/parametru, ci doar preluarea informatiei → **referinta constanta**
 - operatorii aritmetici sau conditionali nu schimba valorile parametrilor;
 - daca este functie membra, atunci va fi functie const;
 - operatorii compusi +=, -= etc. modifica primul argument → **referinta neconstanta**
2. Tipul returnat depinde de sensul operatorului:
 - daca scopul operatorului este de a produce o noua valoare → **const obiect nou**
 - expl: **Integer::operator+** trebuie sa produca un **Integer**
 - rezultatul este const pentru a nu putea fi modificat ca lvalue;
3. Operatorii de atribuire modifica lvalue → **referinta neconstanta**
4. Operatorii logici → **int / bool**



Supraincercarea operatorilor

Return optimizat (*return value optimization*)

`return Integer(left.i + right.i);` **vs** `Integer temp(left.i + right.i);`
`return temp;`

Returnarea directa a unui
temporar:

1. Obiectul returnat se
construieste direct la
adresa de intoarcere a
functiei → NU copy
constructor si NU
destructor pentru obiectul
local

Pasi:

1. Crearea obiectului local temp
→ constructor
2. Copierea acestuia la adresa de
intoarcere a functiei → copy
constructor
3. Distrugerea obiectului temporar



Supraincercarea operatorilor

Conversia automata – prin constructor

Definirea unui constructor cu un parametru de alt tip, permite compilatorului sa faca “automatic type conversion”

```
// Type conversion constructor
class One {
public:
    One() {}
};

class Two {
public:
    Two(const One&) {}
};

void f(Two) {}

int main() {
    One one;
    f(one); // Wants a Two, has a One
} ///:~
```



Supraincarcarea operatorilor

Conversia automata – prin constructor

Daca conversia automata pune probleme → explicit

```
// Using the "explicit" keyword
```

```
class One {  
public:  
    One() {}  
};
```

f(Two(one)) creaza un obiect temporar de tip **Two** din **one**

```
class Two {  
public:  
    explicit Two(const One&) {}  
};
```

```
void f(Two) {}
```

```
int main() {  
    One one;  
    /// f(one); // No auto conversion allowed  
    f(Two(one)); // OK -- user performs conversion  
} ///:~
```



Supraincercarea operatorilor

Conversia automata – prin operator de cast

```
class Three {
    int i;
public:
    Three(int ii = 0, int = 0) : i(ii) {}
};

class Four {
    int x;
public:
    Four(int xx) : x(xx) {}
    operator Three() const { return Three(x); }
};

void g(Three) {}

int main() {
    Four four(1);
    g(four);
    g(1); // Calls Three(1,0)
} ///:~
```



Supraincercarea operatorilor

Supraincercarea operatorului de conversie (cast)

Exemplu:

```
int main()
{
    double x = 1.23;
    int i = (int)x; // conversie catre int ---> parte intreaga
    cout << "i = " << i << endl;
}
```



Supraincercarea operatorilor

Supraincercarea operatorului de conversie (cast)

Sintaxa: **operator tip()**

Exemplu - conversie catre un tip predefinit

```
class Test
{
    int v[2];
    public:
    Test(){v[0] = 100; v[1] = 200;}
    operator int();
};
```

```
Test::operator int ()
{
    return v[0]+v[1];
}
```

```
int main()
{
    Test obiect;
    int x = obiect;
    cout<<x;
    return 0;
}
```




Supraincercarea operatorilor

Supraincercarea operatorului de conversie (cast)

Exemplu - conversie catre un tip definit de utilizator

```
class Test {  
    public:  
    Test(){}  
};
```

```
class Test2 {  
    public:  
    operator Test();  
};
```

```
Test2::operator Test ()  
{  
    return Test();  
}
```

```
void afisare(Test t)  
{cout<<"dupa conversie";}
```

```
int main()  
{  
    Test2 x;  
    afisare(x);  
    return 0;  
}
```



Supraincercarea operatorilor

Supraincercarea unor operatori speciali: “ [] “

- nu poate fi supraincarcat ca functie friend;
- este considerat un operator binar, sub forma operator [] ();
- obiect [5] \Leftrightarrow obiect.operator [] (5);
- sintaxa:

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



Supraincercarea operatorilor

Supraincercarea unor operatori speciali: “ [] “

```
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;  
}
```



Supraincercarea operatorilor

Supraincercarea unor operatori speciali: “ [] “

Daca trebuie sa fie la stanga unei atribuirii, atunci operatorul trebuie sa intoarca referinta.

```
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;  
}
```



Supraincercarea operatorilor

Supraincercarea unor operatori speciali: “ , “

```
#include <iostream>

using namespace std;
class After {
public:
    const After& operator,(const After&) const {
        cout << "After::operator,()" << endl;
        return *this;
    }
};

class Before {};

Before& operator,(int, Before& b) {
    cout << "Before::operator,()" << endl;
    return b;
}

int main() {
    After a, b;
    a, b; // Operator comma called

    Before c;
    1, c; // Operator comma called
}
```

- Nu se foloseste foarte des;

- “for consistency”



Supraincercarea operatorilor

Supraincercarea unor operatori speciali: “ \rightarrow ”

- Cand se doreste ca un obiect sa se comporte ca un pointer (“*smart pointer*”)
- functie membra, nestatica
- operator unar
- obiect->element // obiectul genereaza apelul
- element trebuie sa fie accesibil
- intoarce un pointer catre un obiect din clasa
- utilizat in special in cazul iteratorilor (mai tarziu)



Supraincercarea operatorilor

Supraincercarea unor operatori speciali: “ → “

```
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;  
}
```



Supraincercarea operatorilor

Supraincercarea unor operatori speciali: “ → “

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

class Obj {
    static int i, j;
public:
    void f() const { cout << i++ << endl; }
    void g() const { cout << j++ << endl; }
};

// Static member definitions:
int Obj::i = 47;
int Obj::j = 11;

// Container:
class ObjContainer {
    vector<Obj*> a;
public:
    void add(Obj* obj) { a.push_back(obj); }
    friend class SmartPointer;
};
```

```
class SmartPointer {
    ObjContainer& oc;
    int index;
public:
    SmartPointer(ObjContainer& objc) : oc(objc) { index = 0; }
    // Return value indicates end of list:
    bool operator++() { // Prefix
        if(index >= oc.a.size()) return false;
        if(oc.a[++index] == 0) return false;
        return true;
    }
    bool operator++(int) { // Postfix
        return operator++(); // Use prefix version
    }
    Obj* operator->() const {
        return oc.a[index];
    }
};

int main() {
    const int sz = 10;
    Obj o[sz];
    ObjContainer oc;
    for(int i = 0; i < sz; i++)
        oc.add(&o[i]); // Fill it up
    SmartPointer sp(oc); // Create an iterator
    do {
        sp->f(); // Pointer dereference operator call
        sp->g();
    } while(sp++);
} ///:~
```




Supraincercarea operatorilor

Supraincercarea unor operatori speciali: “ → “

Clasa **Obj** definteste tipul obiectelor utilizate

f() si **g()** printeaza valori folosind date statice

Pointeri catre obiecte → in **ObjContainer**

SmartPointer e clasa prieten si are acces la datele din container

Deplasarea se face utilizand **operator++** (nu se depaseste size)

Nu exista un scop genera pentru **SmartPointer** , e creat special pentru **ObjContainer**



Supraincercarea operatorilor

Supraincercarea unor operatori speciali: “ () “

- nu se creaza un nou mod de apel de functie ci se defineste un mod de a chema functii cu numar variabil de parametri.

Exemplu:

```
double operator( )(int a, float f, char *s);
```

```
Obiect(10, 23.34, "hi");
```

echivalent cu `Obiect.operator()(10, 23.34, "hi");`



Supraincercarea operatorilor

Supraincercarea unor operatori speciali: “ () “
- functie membra, nestatica

```
class loc {  
    int longitude, latitude;  
    public:  
    .....  
    loc operator( )(int i, int j);  
};
```

```
// Overload ( ) for loc.  
loc loc::operator()(int i, int j)  
{  
    longitude = i;  
    latitude = j;  
    return *this;  
}
```

```
//Apel loc ob1(10, 20);  
ob1(7,8); // se executa de sine statator
```



Supraincercarea operatorilor

Supraincercarea <<, >>

Important ca operatorii sa
returneze referinta, pentru a
afecta alte obiecte

```
class IntArray {
    enum { sz = 5 };
    int i[sz];
public:
    IntArray() { memset(i, 0, sz* sizeof(*i)); }
    int& operator[](int x) {
        require(x >= 0 && x < sz,
            "IntArray::operator[] out of range");
        return i[x];
    }
    friend ostream&
        operator<<(ostream& os, const IntArray& ia);
    friend istream&
        operator>>(istream& is, IntArray& ia);
};

ostream&
operator<<(ostream& os, const IntArray& ia) {
    for(int j = 0; j < ia.sz; j++) {
        os << ia.i[j];
        if(j != ia.sz -1)
            os << ", ";
    }
    os << endl;
    return os;
}

istream& operator>>(istream& is, IntArray& ia){
    for(int j = 0; j < ia.sz; j++)
        is >> ia.i[j];
    return is;
}
```



Supraincercarea operatorilor

Supraincercarea altor operatori NEW si DELETE

- supraincercare op. de folosire memorie in mod dinamic pentru cazuri speciale
- size_t: predefinit
- pentru new: constructorul este chemat automat
- pentru delete: destructorul este chemat automat
- supraincercare la nivel de clasa sau globala



Supraincercarea operatorilor

Supraincercarea altor operatori NEW si DELETE

```
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. */
}
```



Supraincercarea operatorilor

Supraincercarea altor operatori NEW si DELETE

```
class loc {  
    int longitude, latitude;  
  
public:  
    loc() {}  
    loc(int lg, int lt) {longitude = lg; latitude = lt;}  
    void show() { cout << longitude << " ";  
                cout << latitude << "\n";}  
    void *operator new(size_t size);  
    void operator delete(void *p);  
};
```

```
// new overloaded relative to loc.  
void *loc::operator new(size_t size){  
    void *p;  
    cout << "In overloaded new.\n";  
    p = malloc(size);  
    if(!p) { bad_alloc ba; throw ba; }  
    return p;  
}
```



Supraincercarea operatorilor

Supraincercarea altor operatori NEW si DELETE

```
class loc {  
    int longitude, latitude;  
  
public:  
    loc() {}  
    loc(int lg, int lt) {longitude = lg; latitude = lt;}  
    void show() { cout << longitude << " ";  
                cout << latitude << "\n";}  
    void *operator new(size_t size);  
    void operator delete(void *p);  
};
```

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




Supraincercarea operatorilor

Supraincercarea altor operatori NEW si DELETE

```
class loc {  
    int longitude, latitude;  
public:  
    loc() {}  
    loc(int lg, int lt) {longitude = lg; latitude = lt;}  
    void show() { cout << longitude << " ";  
                 cout << latitude << "\n";}  
    void *operator new(size_t size);  
    void operator delete(void *p);  
};
```

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

```
int main(){  
    loc *p1, *p2;  
    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;}  
    p1->show();  
    p2->show();  
    delete p1;  
    delete p2;  
    return 0;  
}
```



Supraincercarea operatorilor

Supraincercarea altor operatori NEW si DELETE

- 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



Supraincercarea operatorilor

Cum alegem modul de supraincercare a operatorilor?

Sursa: Rob Murray, C++ Strategies & Tactics, Addison-Wesley, 1993, page 47

Operator	Recomandare de implementare
Toti operatorii unari	Membru
=, [], (), ->, ->*	TREBUIE SA FIE MEMBRE
+=, -=, *=, /=, %=, ^=, &=, =, >>=, <<=	Membru
Toti operatorii binari	Ne - membru



Pointeri catre obiecte

- putem avea Pointeri catre obiecte la fel cum putem avea pointeri catre alte tipuri de variabile.
- retin adresa unui obiect
- acceseaza campurile obiectului folosind operatorul “->”

Expl:

```
class cl {  
    int i;  
public:  
    cl(int j) { i=j; }  
    int get_i() { return i; }  
};  
int main() {  
    cl ob(88);  
    cl *p = &ob; // get address of ob  
    cout << p->get_i();  
    return 0; }
```



Pointeri catre obiecte

- aritmetica pointerilor - relativa la tipul de baza al acestora

Expl:

```
class cl {  
    int i;  
public:  
    cl(int j) { i=j; }  
    int get_i() { return i; }  
};  
int main() {  
    cl ob[2] = {10,20} ;  
    cl *p = ob; p++;  
    cout << p->get_i(); // afiseaza 20  
    return 0; }
```



Pointeri catre obiecte

Pointeri de verificare a tipului in C++

In C++ se poate atribui un pointer altui pointer, doar daca tipurile lor de baza sunt compatibile.

Exemplu:

```
int *p;
```

```
float *q;
```

```
p=q; //eroare, nepotrivire de tipuri
```

Eliminare eroare: - schimbarea de tip (type casting) dar nu se mai aplica verificarile de tip automate facute de C++



Pointeri catre obiecte

Pointeri catre tipuri derivate

In general, un pointer de un anumit tip nu poate indica spre un obiect de tip diferit. Exceptie - clasele derivate, pentru ca functioneaza ca si clasa de baza plus alte detalii.

Utilizati in polimorfism la executie (functii virtuale).

```
class Baza{ public: int x;};  
class Derivata: public Baza {public: int y;};
```

```
int main()  
{  
    Baza *b;  
    Derivata d;  
    b = &d;  
    cout<<b->x; // acces la membrul importat din baza  
    cout<<b->y; // nu are acces pentru ca membrul este propriu derivatei  
    return 0;  
}
```



Pointeri catre obiecte

Pointeri catre tipuri derivate

Rezolvare - conversie de tip

```
class Baza{ public: int x;};  
class Derivata: public Baza {public: int y;};
```

```
int main()  
{  
    Baza *b;  
    Derivata d;  
    b = &d;  
    cout<<b->x; // acces la membrul importat din baza  
    cout<<((Derivata*)b)->y;  
    return 0;  
}
```




Pointeri catre obiecte

Pointeri catre tipuri derivate

Un pointer catre clasa de baza, chiar daca indica spre derivata, la incrementare cauta un alt obiect de tip baza

```
class Baza{ public: int x; } };  
class Derivata: public Baza { };
```

```
int main()  
{  
    Baza *b;  
    Derivata d[2];  
    b = d;  
    d[0].x = 10; d[1].x = 20;  
    cout<<b->x; // afiseaza 10  
    b++;  
    cout<<b->x; // nu afiseaza 20 ci o valoare gresita  
    return 0;  
}
```



Pointeri catre obiecte

Pointeri catre membrii clasei

Sunt folositi rar.

Au asociat, **pe lângă tipul datei sau funcției, și tipul clasă respectiv.**

Un pointer către un membru al unei clase nu se asociază obiectelor clasei respective, **ci clasei**, el conținând ca informație nu adresa membrului ci **deplasarea lui în cadrul clasei.**

Sintaxa:

tip nume_clasa::*pointer_membru;

Atribuirea:

pointer_membru=&nume_clasa::membru;



Pointeri catre obiecte

Pointeri catre membrii clasei

Operatorii destinați accesului la un membru prin pointeri sunt:

- operatorul `.*` dacă se specifică un obiect al clasei (in loc de `.`)
- operatorul `->*` dacă se specifică un pointer de obiect (in loc de `->`)

Sintaxa folosită pentru referirea membrului este:

obiect.*pointer_membru

sau

pointer_obiect->*pointer membru

În cazul pointerilor la funcții membre nu este obligatorie utilizarea operatorului `&`, deci declarațiile:

p_func=&pozitie::deplasare; și p_func=pozitie::deplasare;
sunt echivalente.



Pointeri catre obiecte

Pointeri catre membrii clasei

```
class cl { public:  
    cl(int i) { val=i; }  
    int val;  
    int double_val() { return val+val; } };
```

```
int main() {  
    int cl::*data; // data member pointer  
    int (cl::*func)(); // function member pointer  
    cl ob1(1), ob2(2); // create objects  
    data = &cl::val; // get offset of val  
    func = &cl::double_val; // get offset of double_val()
```

```
    cout << "Here are values: "; cout << ob1.*data << " " << ob2.*data << "\n";  
    cout << "Here they are doubled: ";  
    cout << (ob1.*func)() << " " << (ob2.*func)() << "\n";  
    return 0; }
```



Referinte la obiecte

- in functii: transmiterea prin referinta nu creaza noi obiecte temporare
- se lucreaza direct pe obiectul transmis
- copy-constructorul si destructorul nu mai sunt apelate.

Atentie: obiectul referit poate fi modificat, deci, ideal, referinte constante.

- Intoarcerea din functie a unei referinte.
- permite atribuirea catre apel de functie



Referinte la obiecte

Referinte catre clasele derivate

- putem avea referinte definite catre clasa de baza si apelata functia cu un obiect din clasa derivata;
- exact la la pointeri

```
class Baza{ public: int x;  
void functie() {cout<<"Baza";}};
```

```
class Derivata: public Baza {public: int y;};
```

```
int main()  
{  
    Derivata d;  
    Baza &b = d;  
    b.functie();  
    return 0;  
}
```



Membrii constanți ai unei clase in C++

“The concept of constant (expressed by the const keyword) was created to allow the programmer to draw a line between what changes and what doesn’t.” (Bruce Eckel)

```
#define MAX 100
```

- procesor, nu ocupa memorie, poate fi introdusa intr-un header pentru ca e aceeași pentru tot programul
- nu face verificare de tip, poate introduce probleme.

```
const int MAX = 100;
```



Membrii constanți ai unei clase in C++

```
/// Using const for safety
```

```
#include <iostream>
```

```
using namespace std;
```

```
    const int i = 100; // Typical constant
```

```
    const int j = i + 10; // Value from const expr
```

```
    long address = (long)&j; // Forces storage
```

```
    char buf[j + 10]; // Still a const expression
```

```
int main()
```

```
{
```

```
    cout << "type a character & CR:";
```

```
        const char c = cin.get(); // Can't change
```

```
        const char c2 = c + 'a';
```

```
    cout << c2;
```

```
    // ...
```

```
}
```




Membrii constanți ai unei clase in C++

Pointerii si const

The const specifier binds to the thing it is “closest to.”

Pointer la valori const

const int * u; sau **int const * u;** // u este un pointer care indica catre o constanta intreaga

Pointeri constanti

int d = 1;

int* const w = &d; // w este o constanta de tip pointer care retine doar adresa lui d.

Pointeri constanti la valori constante

int d = 1;

const int* const x = &d; // (1)

int const* const x2 = &d; // (2)



Membrii constanți ai unei clase in C++

Pointerii si const

Atribuirea si verificarea tipului

```
int d = 1;  
const int e = 2;
```

- permisa asignarea unei adrese a unui non const la un pointer const
- NU este permisa asignarea unei adrese a unei constante unui pointer neconst, intrucat compilatorul "intelege" ca s-ar putea modifica valoarea respectiva printr-un pointer.

```
int* u = &d; // OK -- d not const
```

```
//! int* v = &e; // Illegal -- e const
```

```
int* w = (int*)&e; // Legal but bad practice
```



Membrii constanți ai unei clase in C++

Argumentele functiilor si returnarea valorilor

Transmiterea prin parametru constant

```
void f1(const int i) {  
    i++; // Illegal -- compile-time error  
}
```

- se “promite” nemodificarea valorii transmise
- argument transmis fara referinta, deci se face o copie, se incearca incrementarea copiei, deci valoarea transmisa nu se modifica

Mai elegant

```
void f2(int ic) {  
    const int& i = ic;  
    i++; // Illegal -- compile-time error  
}
```



Membrii constanți ai unei clase in C++

Argumentele functiilor si returnarea valorilor

Returnarea unei valori constante

- nu se vad diferente daca se aplica tipurilor predefinite

```
// Returning consts by value
// has no meaning for built-in types
int f3() { return 1; }
const int f4() { return 1; }

int main() {
    const int j = f3(); // Works fine
    int k = f4(); // But this works fine too!
}
```

- recomandata la returnarea tipurilor definite de utilizator; daca se returneaza cu “const”, atunci nu poate fi “l-value”



Membrii constanți ai unei clase in C++

Argumentele functiilor si returnarea valorilor

```
class X { int i;
```

```
public:
```

```
    X(int ii = 0){ i = ii; }
```

```
    void modify(){ i++; }
```

```
};
```

```
X f5() { return X();}
```

```
const X f6() { return X();}
```

```
void f7(X& x) { x.modify(); /* Pass by non-const reference */}
```

```
int main() {
```

```
    f5() = X(1); // OK -- non-const return value
```

```
    f5().modify(); // OK
```

```
// Causes compile-time errors:
```

```
//! f7(f5());
```

```
//! f6() = X(1);
```

```
//! f6().modify();
```

```
//! f7(f6());}
```



Perspective

Cursul 6:

1 Tratarea excepțiilor in C++.

2. Proiectarea descendentă a claselor. Mostenirea in C++.

- Controlul accesului la clasa de bază.
- Constructori, destructori și moștenire.
- Redefinirea membrilor unei clase de bază într-o clasă derivată.
- Declarații de acces.