4 aprilie 2018

POO

C6

Date transmise

Transferul parametrilor

→ prin valoare (recomandat pt variabile care nu sunt obiecte)

→ prin referință → prin ref constantă (⇔ transf prin val dar mai rapid)

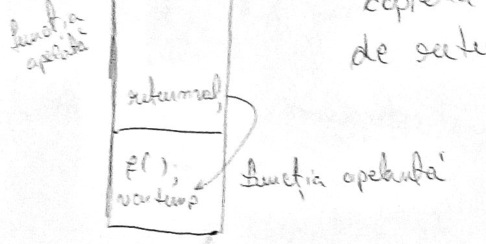
↘ prin ref propriu-zisă (cea pe care o știam)

• constantele de limbaj (nefiind var nu au adresă deci ele sunt și acum nu pot fi transmise prin referință)

Obiectul implicit se transmite metodei prin referință (neconstantă)
(↳ this-ul)

Întoarcerea rezultatului unei funcții

→ prin valoare: crearea unei variabile (obiect) temporare de tipul funcției apelate în funcția apelantă (în zona de memorie ocupată de aceasta pe stivă) și copierea în var temporară a valorii definite de return



const de copiere
destructor

→ prin referință: crearea unei referințe temporare de tipul funcției apelate în funcția apelantă către variabila întoarsă de funcția apelată prin return
↳ prin referință constantă

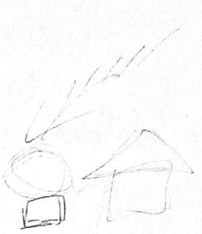
Regulă generală! Variabila întoarsă prin referință trebuie să supraviețuiască funcției apelate

Exemple (standard)   Dacă supraviețuiește funcției poate fi întoarsă, altfel nu

Ce se poate întoarce prin referință	Ce **NU** se poate întoarce prin referință
⊛ variabile alocate dinamic în funcție (care nu trebuie distruse manual) (compilatorul nu lasă să ~~[illegible]~~ întoarcem ca ref alias și dacă nu delete; e responsab. noastră)	⊛ variabilele locale (automatice) (+ param transmiși prin valoare)
	⊛ constantele de limbaj (ele nici nu pot fi nici transferate nici întoarse, că nu-s variabile deci nu le putem asocia o adresă)
⊛ parametrii transmiși prin referință (de e const trb prin ref const)	
⊛ variabilele globale	
⊛ var. locală statică (creată la prima apelare a funcției după aceea își păstrează locația (adresa din mem) până la încheierea programului)	

ref nu const că vrem modif la citire

```
const istream& operator>>(const istream& i, complex& x)
complex a;
int i;
cin>>a>>i;
} i>>x.Re>>x.Im;
return i;
}
```



```
class Complex
{ double Re, Im;
public:
   friend const istream& operator>>(const istream&, complex&);
   double get_Re() {return Re;}
   double get_Im() {return Im;}
};
```

friend → pt când vrem să modif câmpuri private dintr-o funcție externă

Get } acces la câmpuri private, dar fără să le modif
Set → modifică conținutul

ex: la ~~cout~~ << putem să nu definim ca friend ci să apelăm getere le

• consti de copiere
- transmiterea parametrilor } prin valoare
- întoarcerea rezultatului }
- ca consti. de inițializare

complex a, b(a), c = b, d = 5.1;

consti cu param double

consti de copiere (nu e op de atribuire)

c = b; (instrucțiune, op de atribuire)

Facem recuperare joi după vacanță

Conversia datelor

Tipuri tari

→ Conversii
- predefinite
  - I fără pierdere de informații (ex: int → double)
  - II cu pierdere de info (ex: double → int)
- definite de utilizator III

class A; // A poate fi și un tip de date elementar, nu neapărat clasă

```
class B
{ public: B(A);    //constructor pt conversie A→B
                   //va lucra doar cu ob de tip B care lansează metoda în execuție
  operator A ();   → întoarce o val de tip A
           // numele tipului
  { ...
    return ceva-de-tip-A;
  }
};
```

spre deosebire de ceilalți operatori nu mai are tipul returnat în fața "operator" pt că deja e menționat în A (e tipul)

→ operator int() { return x; }
  (x = câmpul care e de tipul int)

Conversii între clase

→ de la o clasă (tip) existentă la o clasă nouă

A → B

↳ se face (cu ajutorul lor) printr-un constructor (scris) în clasa B cu parametru A (de obicei referință constantă către A)

B(A)

• Conversiile nu se pot aplica obiectului care apelează metoda

→ în sens invers : supraîncărcăm cast (*)

Conversii ⟨ implicite (alese de compilator)
explicite (alese de utilizator)

Op cast (de conversie)

(tip) nume_var / int (nume_var);

în C/C++

doar în C++
notație abuzivă
apelăm constructorul lui int

(*): → de la clasa nouă la un tip de date existent

B → A

↳ se face prin supraîncărcarea op. cast al tipului A în clasa B