Constructori şi destructori – Exemple

Exemplul 1.

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

class Copac {
  int inalt;
public:
  Copac(int iniInalt);  // Constructor
  Copac(const Copac&); // Ctor copiere
  ~Copac();  // Destructor
  void creste(int c);
  //inline void creste(int c);
  void printinalt();
};

Copac::Copac(int iniInalt) {
  inalt = iniInalt;
  cout<<"Constructor "<< inalt<<endl;
}

Copac::Copac(const Copac& c) {
  inalt = c.inalt;
  cout<<"Constructor copiere "<< inalt<<endl;
}

Copac::~Copac() {
  cout << "in destructor  Copac " ;
  printinalt();
}

inline void Copac::creste(int c) {
  inalt += c;
}

void Copac::printinalt() {
  cout << "inalt copac  = " << inalt << endl;
}
```

```
Copac cg(0);  // GLOBAL

int main()
{
  cout<<"Start main"<<endl;
//  Copac tt; Cannot generate default constructor
//  since constructors were declared
  Copac c0(1); // constructor cu parametri
  Copac c1(c0); // constructor de copiere
  Copac c2=cg; //// ctor de copiere nu atribuire!!!
// forma invechita a ctor-ului de copiere
  Copac *pc=new Copac(2);
  for(int i=0;i<2;i++)
  {
    cout << "local {" << endl;
    Copac cl(10);
    cout << "dupa creare Copac " ;
    cl.printinalt();
    cl.creste(5);
    cl.printinalt();
    static Copac cs(100);  // STATIC
    cout << "inainte de }" << endl;
  }
  cout << "dupa }" << endl;
  delete pc; // fara, nu se elibereaza pc!
  cout<<"Stop main"<<endl;
  return 0;
}
```

Rezultatul executiei :

```
Constructor 0
Start main
Constructor 1
Constructor copiere 1
Constructor copiere 0
Constructor 2
local {
Constructor 10
dupa creare Copac inalt copac  = 10
inalt copac  = 15
Constructor 100
inainte de }
in destructor  Copac inalt copac  = 15
local {
Constructor 10
dupa creare Copac inalt copac  = 10
inalt copac  = 15
inainte de }
in destructor  Copac inalt copac  = 15
dupa }
in destructor  Copac inalt copac  = 2
Stop main
in destructor  Copac inalt copac  = 0
in destructor  Copac inalt copac  = 1
in destructor  Copac inalt copac  = 1
in destructor  Copac inalt copac  = 100
in destructor  Copac inalt copac  = 0
```

Exemplul 2.

```
#include <iostream>
#include <string.h>
using namespace std;
class Sir
{
   char *psir;
   int lung;
  public:
   Sir(char * s="");
   Sir(int);
   Sir(const Sir&);
   void afissir();
   ~Sir(){ delete []psir;}
};

Sir:: Sir (char *sursa)
{
   lung=strlen(sursa);
   psir=new char[lung+1];
   strcpy(psir, sursa);
   cout<<"Constructor
sursa="<<sursa<<endl;
}

Sir:: Sir(int l)
{
   lung=l;
   psir=new char[l+1];
        cout<<"Constructor sursa="<<l<<endl;
   *psir='\0';
}

Sir ::Sir(const Sir& sursa)
{
   lung=sursa.lung;
   psir=new char[lung+1];
   strcpy(psir, sursa.psir);
   cout<<"Constructor cop. sursa="
        <<sursa.psir<<endl;
}

void Sir::afissir()
{
   cout<<psir;
}

int main()
{
   Sir s1(70), s2("OK");
   cout<<"s1=";
   s1.afissir();
   cout<<endl;
   cout<<"s2=";
   s2.afissir();
   cout<<endl;
   Sir s3("C++ este un C mai bun");
   cout<<"s3=";
   s3.afissir();
   cout<<endl;
   Sir s4(s3);
   cout<<"s4=";
   s4.afissir();
   cout<<endl;
   return 0;
}
```

Rezultatul executiei :

```
Constructor sursa=70
Constructor sursa=OK
s1=
s2=OK
Constructor sursa=C++ este un C mai bun
s3=C++ este un C mai bun
Constructor cop. sursa=C++ este un C mai
bun
s4=C++ este un C mai bun
```

Modificatorul *const*

- Modificatorul ***const*** modifică tipul unei date în sensul restrângerii modului său de utilizare
- Datele declarate folosind modificatorul ***const*** nu pot fi modificate în mod direct
- Există următorele situaţii distincte semnficative în care se utilizează modificatorul ***const***
    - Declaraţii de variabile sau instanţieri de obiecte
    - Declaraţii de parametri formali
    - Antete/prototipuri de funcţii, fie pentru a proteja valoarea returnată de funcţie, fie pentru a semnala faptul că funcţia membru nu modifică obiectul ce o apelează (***const*** face parte din semnatura !)

Exemple :

1. Declaraţia variabilelor

```
a) const int i=3;
   const double pi=3.141;
   i=4; i=3; pi=3.1415926; !GRESIT

b) const char *sir=”abc”;
   *sir=’A’; ! GRESIT
   sir++; CORECT

c) char *const sir=”abc”;
   *sir=’A’; CORECT
   sir++; ! GRESIT

d) const char *const sir=”abc”;
   *sir=’A’; GRESIT
   sir++; ! GRESIT

e) const int i=2;
   int *p=&i; ! GRESIT
   const int *p=&i;  CORECT

f) class NC{ ..};
   const NC ob(..); // obiect constant
```

2. Transferul parametrilor

a)
```
char *strcpy(char *dest, const char *sursa);
```

b)
```
class NC
{…
      public:
NC(const NC&);…}
const NC ob1(..);
NC ob2(ob1);  // eroare in absenta lui const!
```

3. Protejarea valorilor returnate de funcţii

a)
```
const char *denluna(int n)
{
      static const char *tpl[]={“ilegal”,”ian”, “feb”..};
      return (n<1||n>12?tpl[0]:tpl[n];
}
```

b)
```
const int j=7; // global
int *func(void)
{
      return &j; // !GRESIT
}
int *pp=func(); *pp=-7; !!!

const int *func(void) // corect
{
      return &j; //
}
int *pp=func(); // !GRESIT
const int *pp=func(); // CORECT
```

4. Specificare faptului ca o funcţie membru nu modifica obiectul pentru care a fost apelată

```
tip_ret NC::f_membru(lista_par) const //const face parte din semnatura

void Sir::afissir() const;

const Sir s(“Nu ma modifica!”);
s.affisir(); // fara const NU se putea apela!
```

Funcţii şi clase *friend*

- Pentru a permite accesul unor funcţii ce nu sunt metode ale unei clase la datele membru protejate ale clasei s-a definit conceptul de funcţii ***friend***
- Prototipul unei funcţii friend se scrie astfel :

```
class NumeClasa
{
 ……..
        friend  tip_returnat  nume_functie_friend(lista_parametri);
……..
};
```

- Unei funcţii ***friend*** nu i se transmite pointerul ***this***
- Atunci când se defineşte în interiorul clasei, funcţia este ***inline*** (daca acest lucru este posibil)
- Atunci când se defineşte în afara clasei, numele funcţiei NU se califică folosind operatorul de rezoluţie
- În cazul în care se doreşte ca toate metodele unei clase să fie funcţii ***friend*** pentru o clasă dată, se declară întreaga clasă ca fiind ***friend*** al clasei date

```
class NumeClasa1;
class NumeClasa2
{
 ……..
        friend NumeClasa1;
……..
};
```

- Relaţia ***friend*** nu e tranzitivă

Supraîncărcarea operatorilor în limbajul C++

- Tipurile abstracte de date au fost concepute astfel încât să permită folosirea lor de către utilizator într-o manieră similară tipurilor predefinte. În acest scop se pune la dispoziţia programatorului mecanismul de supraîncărcare a operatorilor
- Limbajul C++ permite supraîncărcarea numai pentru operatorii existenţi (nu se pot crea operatori noi)
- Nu se pot supraîncărca următorii operatori :

  . (punct)
  :: (rezoluţie)
  .*(selecţie a membrilor prin intermediul pointerilor la membri)
  ?: (condiţional)
  sizeof
  typeid

- Nu se pot supraîncărca directivele adresate preprocesorului **#** şi **##**
- Nu se pot crea operatori noi
- Prin supraîncărcare nu se schimbă *n*-aritatea, prioritatea sau asociativitatea operatorilor (acestea rămânând cele predefinite)
- Supraîncărcarea operatorilor se poate face folosind funcţii membru sau funcţii ***friend***
- Sintaxa generală pentru supraîncărcarea operatorilor este :
  - pentru funcţii friend :

    tip_returnat ***operator*** op(lista_parametri) { corp}
  - pentru funcţii membru :

    tip_returnat NumeClasa :: ***operator*** op(lista_parametri) { corp}
- În general, pentru acelaşi operator, funcţiile ***friend*** au un argument în plus în lista de parametri, funcţiile membru având implicit transmis operandul pentru care au fost apelate (prin intermediul pointerului ***this***)

Exemplul 4. Clasa Complex cu operatori supraîncărcaţi

```
#include <iostream >
#include <math.h>
using namespace std;

class Complex
  {
     double real;
     double imag;
    public:

     Complex(double x=0.0, double y=0.0)
     {
        real=x;imag=y;
     }

     Complex(const Complex& c)
     {
        real=c.real;imag=c.imag;
      }

     double mod()
     {
        return sqrt(real*real+imag*imag);
     }

     double& retreal()
     {
        return real;
     }

     double& retimag()
     {
        return imag;
     }

     Complex operator + (const Complex &) const;
     friend  Complex operator + (double , const Complex& );
     friend istream& operator >>(istream &, Complex&);
     friend ostream& operator <<(ostream &, const Complex&);
     ~Complex(){}
  };
```

```
Complex Complex :: operator + (const Complex& z) const
  {
     Complex temp;
     temp.real=real+z.real;
     temp.imag=imag+z.imag;
     return temp;
  }

  Complex operator + (double x, const Complex& z)
  {
     Complex temp;
     temp.real=x+z.real;
     temp.imag=z.imag;
     return temp;
  }

  istream& operator >>(istream & in, Complex& z)
 {
        in>>z.real>>z.imag;
        return in;
 }

 ostream& operator <<(ostream &out, const Complex& z)
 {
        out<<z.real<<"+i"<<z.imag<<endl;
        return out;
}

int main(void)
{
                Complex z1;
                const Complex z4(2,2);
                Complex z5;
                z1=z4+z5;
                cout<<z1<<endl;
                cout<<"z1=";
                cin>>z1;
                cout<<"z1= "<<z1;
                z1=3.0+z4;
                cout<<"z1= "<<z1;
                return 0;
}
```