Laborator 1-2 - OOP

Initializarea variabilelor intr-un program CPP:

```
#include <iostream.h>
#include <string.h>

/* initializare tablou cu 5 elemente numere intregi */
int a[5] = {1, 2, 3, 4, 5};
/* primul element se initializeaza cu 1; celelalte elemente se initializeaza cu
0 */
int b[3] = {1};
/* dimensiunea vectorului este egala cu numarul datelor */
int c[] = {4, 3, 2, 1};

/* definitia unei structuri*/
struct A{
  int i;
  float f;
  char c;
};


/* initializare atribute structura */
A va = {1, 2.32, 'x'};

/* initializare tablou de structuri */
struct A tab[]={{1, 1.1, 'a'}, {2, 2.2, 'b'}, {3, 3.3, 'c'}};

class B{
      public:
            int i;
            char s[10];

            void f(){
              i = i + 1;
            }
};

/* initializare atribute publice ale unui obiect */
B vb = {1, "alocare"};

class C{
            int i, j;
      public:
            C(int k = 0, int l = 0){
              i = k;
              j = l;
            }
};

void main(){
  /* initializare tablou de obiecte; se apeleaza constructorul definit */
  C tabc[] = { C(1,2) , C(3,4) , C(5,6)};
}
```

---------- Exemplul 1: Alocare dinamica si tratarea erorilor ----------

```
#include <iostream.h>
#include <new.h>
#include <stdlib.h>

#define SIZE 1000

class TestA{
```

```
            /* tablou de elemente intregi */
            int tab[SIZE];
            /* membru static; pastreaza numarul de obiecte construite*/
            static int index;
      public:
            TestA(){
              cout<<index++<<" ";
            }
};

/* initializare membru static */
int TestA::index = 0;

/* functie pentru tratarea situatiei cand memoria
   libera devine insuficienta fata de memoria necesara */
void mem_error(){
  cerr<<"Memorie insuficienta!"<<endl;
  exit(1);
}

void main(){
  //initializarea "handler"-ului pentru tratarea erorilor de alocare
  set_new_handler(mem_error);
  cout<<endl;

  while (1){
    new TestA[3];
  }
}
```

---------- Exemplul 2: Supraincarcarea operatorului ‘new’ local ----------

```
#include <iostream.h>
#include <stdlib.h>

class TestB{
            int x;
      public:
            /* supraincarcarea operatorului 'new'
               corespunzator clasei TestB */
            void* operator new(size_t dim){
              cout<<"Operatorul new "<<dim<<endl;
              return malloc(dim);
            };
            /* supraincarcarea operatorului 'new[]'
               corespunzator clasei TestB.
               BC 3.1 nu accepta supraincarcarea acestui
               operator decat ca functie globala
            void* operator new[](size_t dim){
              cout<<"Operatorul new[] "<<dim<<endl;
              return malloc(dim);
            };*/
};

void main(){
  TestB *unu = new TestB;
  TestB *multi = new TestB[100];

  delete unu;
  delete[] multi;
}
```

---------- Exemplul 3: Supraincarcarea operatorului new global ----------

```
#include <stdlib.h>
#include <stdio.h>

/* supraincarcarea operatorului 'new' global */
void* operator new(size_t dim){
  printf("Operatorul new global %d\n",dim);
  return malloc(dim);
}

/* supraincarcarea operatorului 'delete' global */
void operator delete(void *p){
  printf("Operatorul delete global %p\n",p);
  free(p);
}

class TestC{
            int x;
      public:
            /* supraincarcarea operatorului 'new' local,
               corespunzator clasei TestC */
            void* operator new(size_t dim){
              printf("Operatorul new %d\n",dim);
              return malloc(dim);
            };

            /* supraincarcarea operatorului 'delete' local,
               corespunzator clasei TestC */
            void operator delete(void *p){
              printf("Operatorul delete %p\n",p);
              free(p);
            };
};

void main(){
  //se apeleaza operatorul 'new' din cadrul clasei TestC
  TestC *unu = new TestC;
  //se apeleaza operatorul 'new' global
  TestC *multi = new TestC[100];

  //se apeleaza operatorul 'delete' definit in cadrul clasei TestC
  delete unu;
  //se apeleaza 'delete' global
  delete[] multi;
}
```

---------- Exemplul 4: Supraincarcarea operatorului ‘new’ particularizat ----------

```
#include <iostream.h>
#include <stdlib.h>

class TestD{
            /* atribut privat de tip intreg */
            int x;
            /* membru static; pastreaza numarul de obiecte construite*/
            static int nr;
      public:
            /* Operatorul new supraincarcat.
               - dim    dimensiunea zonei de memorie
               - p      adresa zonei de memorie de unde se va realiza alocarea
            */
            void* operator new(size_t dim, void *p){
```

```
              cout << "Operatorul new " << dim << endl;

              return &((int *)p)[nr++];
            };
};

/* initializare membru static */
int TestD::nr = 0;

void main(){
  int x[1000];
  //se apeleaza operatorul 'new' supraincarcat in clasa TestD
  TestD *unu = new(x) TestD;

  delete unu;
}
```

---------- Clase container si iterator ----------

```
#include <iostream.h>

/* Clasa Stack implementeaza o stiva alocata 'static' ce pastreaza
numai elemente de tip intreg. */
class Stack {
            /* size joaca rol de constanta */
            enum { size = 100 };
            /* tablou alocat static cu elemente numere intregi
               in care se vor pastra elementele stivei */
            int stack[size];
            /* varful stivei */
            int top;
      public:
            /* Constructor fara parametrii. Initializeaza varful pe 0.*/
            Stack() : top(0) {
              stack[top] = 0;
            }

            /* Metoda de adaugare a unui element in stiva:
               se va adauga un element daca stiva nu este plina.
               - i - valoarea elementului care se adauga
            */
            void push(int i){
              if (top < size)
                stack[top++] = i;
            }

            /* Metoda de extragere a unui element din stiva.
               - return - elementul din varful stivei.
               Daca stiva este vida, intoarce tot ultimul element
               (cel mai vechi element nu va fi 'extras' niciodata).
            */
            int pop(){
              return stack[(top > 0) ? --top : top];
            }

            /* Clasa declarata clasa prietena pentru a avea acces
            la membrii privati */
            friend class StackIterator;
};

/* Clasa StackIterator implementeaza un 'iterator' = un obiect cu ajutorul
caruia se poate naviga printre obiectele unui 'container'. Un 'container'
este un obiect ce poate pastra referinte catre alte obiecte. */
```

```
class StackIterator {
            /* referinta catre un obiect de tip Stack -
               catre stiva ce este parcursa */
            Stack& s;
            /* indicele elementului curent din cadrul tabloului stack
               apartinand unui obiect de tip Stack */
            int index;
      public:
            /* Constructor obisnuit.
               -_s - referinta catre obiectul de tip Stack
            */
            StackIterator(Stack& _s) : s(_s), index(0) {}

            /* Operatorul ++ (preincrementare) supraincarcat: realizeaza
            trecerea la elementul urmator din stiva.*/
            int operator++ (){
              if (index < s.top - 1)
                index++;

              return s.stack[index];
            }

            /* Operatorul ++ (postincrementare) supraincarcat: realizeaza
            trecerea la elementul urmator din stiva.*/
            int operator++ (int){
              int returnval = s.stack[index];

              if (index < s.top - 1)
                index++;

              return returnval;
            }
};


/* Functia calculeaza elementele sirului lui Fibonacci;
   acestea sunt salvate intr-un tablou static, ce isi pastreaza
   valorile intre doua apeluri succesive ale functiei fibonacci.
   Functia returneaza valoarea celui de-al n-lea element calculat. */
int fibonacci(int n){
  //constanta max
  const int max = 100;
  /* tablou static (alocat in zona de date) de elemente intregi
     pastreaza valorile intre doua apeluri ale functiei */
  static int f[max];

  /* initializare primele doua numere din sirul lui Fibonacci. */
  f[0] = f[1] = 1;

  int i;
  /* determina indicele primului element din sirul lui Fibonacci
     care nu a fost calculat. */
  for (i = 0; i < max; i++)
    if (f[i] == 0)
      break;

  while (i <= n){
    //calculeaza elementul i dupa formula
    f[i] = f[i - 1] + f[i - 2];
    i++;
  }

  /* intoarce valoarea ceruta */
  return f[n];
```

```
}

void main(void){
  Stack is; //se aloca un obiect de tip Stack

  /* pune pe stiva primele 15 numere din sirul lui Fibonacci. */
  for (int i = 0; i < 15; i++)
    is.push(fibonacci(i));

  //creeaza un iterator
  StackIterator it(is);

  /* parcurge elementele stivei cu ajutorul iteratorului */
  for (int j = 0; j < 15; j++)
    cout << it++ << endl;

  /* "extrage" elementele de pe stiva prin metoda "pop" */
  for (int k = 0; k < 15; k++)
    cout << is.pop() << endl;
}
```

---------- Proiect Poligon ----------

---------- Punct.h ----------

```
#ifndef _PUNCT_
#define _PUNCT_

class Punct{
    int x, y;
  public:
    Punct(int _x = 0, int _y = 0){
      x = _x;
      y = _y;
    }

    void setX(int _x){
      x = _x;
    }

    void setY(int _y){
      y = _y;
    }

    int getX(){
      return x;
    }

    int getY(){
      return y;
    }
};
#endif
```

---------- Vector.h ----------

```
#ifndef _VECTOR_
#define _VECTOR_

#include <stdlib.h>

class Vector{
    void** tab;
    int capacity;
    int nrElemente;
```

```
  public:
    Vector(int _capacity){
      capacity = _capacity;
      tab = (void**) new (void*) [capacity];
      nrElemente = 0;
    }

    ~Vector(){
      delete [] tab;
    }

    int addLast(void* element){
      if (nrElemente < capacity){
            tab[nrElemente] = element;
            nrElemente++;
            return 1;
      }
      return 0;
    }

    int insertElement(void* element, int position){
      if (0 <= position && position < nrElemente){
            tab[position] = element;
            return 1;
      }
      return 0;
    }

    void* getElement(int position){
      if (0 <= position && position < nrElemente)
            return tab[position];
      else
            return NULL;
    }

    int deleteElement(int position);

    int getNumber(void){
      return nrElemente;
    }
};
#endif
```

---------- Vector.cpp ----------

```
#include "vector.h"

int Vector::deleteElement(int position){
  if (0 <= position && position < nrElemente){
    for (int i = position; i < nrElemente - 1; i++)
      tab[i] = tab[i + 1];
    nrElemente--;
    return 1;
  }
  return 0;
}
```

---------- Poligon.h ----------

```
#ifndef _POLIGON_
#define _POLIGON_

#include <iostream.h>
#include <stdlib.h>
#include "vector.h"
```

```
#include "punct.h"
#define MAX 50

class Poligon{
    Vector varfuri;
  public:
    Poligon(Punct* tab, int n);

    Punct* getVarf(int position){
      void* p = varfuri.getElement(position);

      if (p != NULL)
        return (Punct*)p;
      else
        return NULL;
    }

    int addVarfLast(Punct* p){
      return varfuri.addLast(p);
    }

    int addVarf(Punct* p, int position){
      return varfuri.insertElement(p, position);
    }

    int removeVarf(int position){
      return varfuri.deleteElement(position);
    }

   void afis();
};
#endif
```

---------- Poligon.cpp ----------

```
#include "poligon.h"

Poligon::Poligon(Punct* tab, int n) : varfuri(MAX){
  for (int i = 0; i < n; i++)
    varfuri.addLast(&tab[i]);
}

void Poligon::afis(void){
  int n = varfuri.getNumber();

  for (int i = 0; i < n; i++){
    Punct *p = getVarf(i);
    cout << "Varf: " << i << "x: " <<p->getX() << "x: " <<p->getY() << "\n";
  }
}
```

---------- Main.cpp ----------

```
#include "punct.h"
#include "poligon.h"

void main(void){
  Punct p[] = {Punct(1,1), Punct(7,7), Punct(3,3), Punct(4,4)};
  Poligon polig(p, 4);

  polig.addVarfLast(new Punct(5,5));
  polig.addVarf(new Punct(6,6), 2);

  polig.removeVarf(3);
  polig.afis();
}
```