Laborator 2

~funcții și clase prietene, liste de inițializare~

Liste de inițializare (Member initializer lists)

Unele tipuri de date (cum am văzut la seminar referințe, dar adăugăm la această listă acum și const-uri) trebuie să fie inițializate pe aceeași linie cu declararea. Exemplu:

```
class Test
{
private:
   const int _value1;
   double _value2;
   char _value3;
public:
   Test(int value1, double value2, char value3)
   {
       _value1 = value1; // eroare: nu putem asigna unei variabile
constante
       _value2 = value2;
       _value3 = value3;
       //ce facem noi aici pentru _value1 este:
       const int _value1; // eraore: variabilele constante nu pot
fi
       // initalizate fara valoare
       _value1 = 5; // eroare: nu pot face assignment la
       // variabila constanta
   }
};
```

Soluția în acest caz o reprezintă listele de inițializare:

```
class Test
{
private:
   const int _value1;
   double _value2;
   char _value3;
public:
   Test(int value1, double value2, char value3) : _value1(value1),
```

```
_value2(value2), _value3(value3)
   {
       //nu mai tebuie sa facem nimic aici
   }
};
```

Atenție! Lista de inițializare se poate folosi DOAR la constructori.

Dacă nu ne regăsim în unul dintre cazurile de mai sus (avem date memebre const sau referințe), mai folosim liste de inițializare pentru că arată mai clean, dar mai important, pentru eficiență:

```
class example
{
private:
   int _x;
public:
   example()
   {
       std::cout<<"S-a apelat constructorul fara parametri\n";
   }
   example(int x)
   {
       std::cout<<"S-a apelat constructorul parametrizat cu
parametru "<<x<<"\n";
   }

};
class entity
{
private:
   example _example;
public:
   entity()
   {
       _example = example(8); // Ca sa atribuim lui _example
valoarea noului obiect
       // creat, este nevoie ca _example sa fie creat.
       // => Se apeleaza constructorul fara parametri pentru
```

```
        // creearea obiectului _example.
        // Apoi se apeleaza constructorul parametrizat
        // pentru 'example(8)'.
        // Apoi se face atribuirea (assignment).
    }

};

int main()
{
    entity e; // Se afiseaza:
    // S-a apelat constructorul fara parametri
    // S-a apelat constructorul parametrizat cu parametru 8
    return 0;
}
```

În schimb, dacă folosim liste de inițializare, se apelează direct constructorul parametrizat:

```
class example
{
private:
    int _x;
public:
    example()
    {
        std::cout<<"S-a apelat constructorul fara parametri\n";
    }
    example(int x)
    {
        std::cout<<"S-a apelat constructorul parametrizat cu
parametru "<<x<<"\n";
    }

};
class entity
{
private:
    example _example;
public:
    entity() : _example(8) {}
```

```
};

int main()
{
   entity e; // Se afiseaza doar:
   // S-a apelat constructorul parametrizat cu parametru 8
   return 0;
}
```

De reținut:
Ordinea în care se inițializează datele din listele de inițializare nu este ordinea specificată în listele de inițializare, ci este ordinea în care datele respective au fost declarate în clasă. De aceea, pentru a evita confuzii, e bine să scrieți variabilele în lista de inițializare în aceași ordine în care ați scris variabilele în clasă.

Compoziția

Compoziția este un concept fundamental al programării orientate pe obiecte care modelează relațiile de tipul *„has-a”* între obiecte. Asta înseamnă că dacă avem clasa actor și clasa film, putem spune că un film are actori.

```
class point2D
{
   int _x;
   int _y;
public:
   point2D() : _x(0), _y(0) {}
   point2D(int x, int y) : _x(x), _y(y) {}
   void set_point2D (int x, int y)
   {
       _x = x;
       _y = y;
   }

};
class creature
{
   std::string _name;
   point2D _location;
```

```
public:
   creature (const std::string& name, const point2D & location) :
   _name(name),
   _location(location){}
   void move_to(int x, int y)
   {
       _location.set_point2D(x, y);
   }

};
int main()
{
   //creature creature1("troll", (4, 7)); // Aici nu stie ca 4 si
7 sunt pentru
   // initializarea punctului.
   point2D point1(4,7); // Dar pot sa initializez asa.
   creature creature1("troll", point1);
   creature creature2{"bear", {1,1}}; // Sau asa.
   return 0;
}
```

Funcții și clase prietene

Pentru a controla accesul la membrii unei clase, se utilizează specificatori de acces: **public**, **protected** și **private**. În cazul specificatorilor protected și private, memberii respectivi nu pot fi accesați în afara clasei în care sunt declarați. Putem însă încălca această regulă, folosind în cadrul unei clase cuvântul cheie friend, care permite unei funcții externe sau unei alte clase accesul la membri protected și private.

Sintaxa este următoarea:

```
class NumeClasa
{
   //cod
   friend TIP_RETURNAT NUME_FUNCTIE(ARGUMENTE);
   //cod
};
```

sau

```
class AltaClasa
```

```
{
  //cod
};
class NumeClasa
{
   //cod
   friend class AltaClasa;
   //cod
};
```