# Elemente de bază ale limbajului C++
# (Suport pentru înțelegerea elementelor de sintaxă pentru laboratoarele SDA)

- fișierele **.h**

  **-** fișiere de tip antet (*header files*) care conţine declaraţii şi definiţii (de funcţii, de variabile)

OBS: În fișierele de tip antet vor fi declarate variabilele-membru necesare pentru reprezentarea în memorie a tipurilor abstracte de date (TAD-urilor) (în zona *private*), precum și operațiile din interfața TAD-urilor(în zona *public*).

- fișierele **.cpp**

  **-** fișiere text de tip sursă (*source code files*) care conţin cod sursă C++. Fiecare fişier sursă conţine una sau mai multe funcţii şi, eventual, referinţe către unul sau mai multe fişiere de tip antet.

OBS: În fișierele de tip sursă vor fi implementate operațiile expuse în interfața unui TAD (și, deci, declarate în zona *public* în fișierul de tip antet aferent).

- **#include**

  - directivă care permite includerea fișierelor de tip antet (*header*) în codul sursă

Exemplu:

```
1.  #include "Colectie.h"
2.  #include "IteratorColectie.h"
3.  #include <exception>
4.  #include <iostream>
```

Liniile 1 și 2 includ fișiere de tip antet definite de utilizator, pe când liniile 3 și 4 includ fișiere de tip antet definite în biblioteca standard C++.

- **main()**

  - funcția cu denumire fixată care reprezintă programul principal, cu aceasta începând execuția.

- **#pragma once**

  - directivă (non-standard, dar suportată de majoritatea compilatoarelor) care asigură ca un fișierul de tip antet să fie inclus o singură dată de către preprocesor

- **private**

  - modificator de acces, care definește următoarele reguli de acces ale variabilelor membru și metodelor dintr-o clasă:

| | Accesul este permis |
|---|---|
| În interiorul clasei | DA |
| În clasele derivate | NU |
| În exterior | NU |
| În clasele prietene | DA |

OBS: Variabilele membru necesare pentru reprezentarea în memorie a unui TAD se vor declara în zona *private*.

Exemplu:

```
1.  private:
2.      const Colectie& col;
3.      /* aici e reprezentarea specifica a iteratorului*/
```

În exemplul anterior, referința la colecția iterată, necesară reprezentării iteratorului pe colecție, este declarată în zona *private*, nefiind expusă în exterior clasei (exceptând clasele prietene). Alte variabile-membru (corespunzătoare cursorului iteratorului - de pildă, indexul curent) vor fi, de asemenea, declarate în zona *private*.

- **public**

  - modificator de acces, care definește următoarele reguli de acces ale variabilelor membre și metodelor dintr-o clasă:

| | Accesul este permis |
|---|---|
| În interiorul clasei | DA |
| În clasele derivate | DA |
| În exterior | DA |
| În clasele prietene* | DA |

OBS: Operațiile din interfața unui TAD se vor declara în zona *public*.

Exemplu:

```
1.  class Colectie
2.  {
3.      friend class IteratorColectie;
4.
5.  private:
6.      /* aici e reprezentarea */
7.  public:
8.          //constructorul implicit
9.          Colectie();
10.
11.         //adauga un element in colectie
12.         void adauga(TElem e);
13.
14.         //sterge o aparitie a unui element din colectie
15.         //returneaza adevarat daca s-a putut sterge
16.         bool sterge(TElem e);
17.
18.         //verifica daca un element se afla in colectie
19.         bool cauta(TElem elem) const;
20.
21.         //returneaza numar de aparitii ale unui element in colectie
22.         int nrAparitii(TElem elem) const;
23.
24.         //intoarce numarul de elemente din colectie;
```

```
25.         int dim() const;
26.
27.         //verifica daca colectia e vida;
28.         bool vida() const;
29.
30.         //returneaza un iterator pe colectie
31.         IteratorColectie iterator() const;
32.
33.         // destructorul colectiei
34.         ~Colectie();
35.
36.     }
```

În exemplul anterior, toate operațiile care compun interfața TAD-ului colecție au fost declarate în zona *public*, permițându-se ,astfel, accesul lor în afara clasei *Colectie*.

- [nume_clasă]::[nume_funcție]

**-** operator de rezoluție care permite accesul la un identificator cu domeniu fișier, dintr-un bloc în care acesta nu este vizibil, datorita unei alte declarații. Principala aplicație a operatorului de rezoluție este legată de clase.

OBS: În cazul proiectelor SDA, interfețele TAD-urilor sunt declarate în fișierele de tip antet, unde sunt definite clasele, iar funcțiile din interfețe vor fi implementate în fișierele de tip sursă aferente. Astfel, este necesară definirea unei funcții în afara clasei (dar, bineînţeles, în domeniul ei de definiţie). Pentru aceasta, în fișierele de tip sursă (*.cpp*), numele fiecărei funcţii trebuie prefixat de numele clasei prin intermediul operatorului de rezoluţie ::.

Sintaxa de definire a unei funcţii în afara clasei este următoarea:

```
tip_returnat nume_clasa::nume_funcţie(lista_argumente){
        //corpul funcţiei
}
```

Exemplu:

```
1.  Colectie::Colectie() {
2.      /* de adaugat */
3.  }
4.
5.
6.  void Colectie::adauga(TElem elem) {
7.      /* de adaugat */
8.  }
9.
10.
11. bool Colectie::sterge(TElem elem) {
12.     /* de adaugat */
13.     return false;
14. }
15.
16.
17. bool Colectie::cauta(TElem elem) const {
18.     /* de adaugat */
19.     return false;
20. }
```

```
21.
22.  int Colectie::nrAparitii(TElem elem) const {
23.      /* de adaugat */
24.      return 0;
25.  }
26.
27.
28.  int Colectie::dim() const {
29.      /* de adaugat */
30.      return 0;
31.  }
32.
33.
34.  bool Colectie::vida() const {
35.      /* de adaugat */
36.      return true;
37.  }
38.
39.  IteratorColectie Colectie::iterator() const {
40.      return  IteratorColectie(*this);
41.  }
42.
43.
44.  Colectie::~Colectie() {
45.      /* de adaugat */
46.  }
```

Exemplul anterior ilustrează conținutul fișierului *Colecție.cpp ,* în care se vor implementa metodele clasei Colecție, definite în fișierul de tip antet *Colecție.h.* Astfel, toate operațiile sunt precedate de numele clasei (*Colectie*) prin intermediul operatorului de rezoluție (::).

- **typedef**

- cuvânt-cheie care permite declararea unui identificator ca un *alias* pentru un tip de date sau, cu alte cuvinte, asignarea unui nou nume pentru un tip de date existent.

Exemplu:

```
typedef int TElem;
```

În acest exemplu, identificatorul TElem este declarat ca un *alias* pentru tipul de date *int,* ceea ce, în cadrul proiectului, va sugera lucrul cu un tip de date generic.

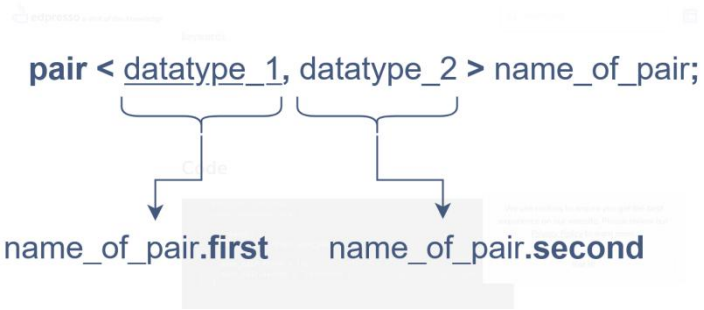
- **std::pair**<T1, T2>

- șablon de clasă, definit în header-ul *<utility>,* care permite stocarea a două obiecte eterogene într-o singură entitate. Este o particularizare a **std::tuple** pentru două elemente.

```
template<
   class T1,
   class T2
> struct pair;
```

T1 și T2 reprezintă tipurile celor două obiecte componente.

Accesarea celor două obiecte componente se face prin cuvintele cheie **first** și **second**.




Exemplu:

```
1.  typedef int TCheie;
2.  typedef int TValoare;
3.
4.  #include <utility>
5.  typedef std::pair<TCheie,TValoare> TElem;
```

În acest exemplu, TElem a fost declarat ca un *alias* pentru o pereche de tipul <*cheie, valoare*>, *cheie* fiind de tipul TCheie (int), iar *valoare* fiind de tipul TValoare (int).

- **#define**

- directivă de preprocesare (recunoscută de compilator prin prezența caracterului #) care permite definirea constantelor simbolice.

Sintaxa pentru definirea unei constante simbolice este:
#define simbol valoare

- aceasta are ca efect înlocuirea tuturor aparițiilor lui *simbol* în codul sursa (cu excepția aparițiilor în cadrul unor constante de tip șir, în comentarii sau în componența unui alt identificator) cu *valoare* înaintea compilării codului sursă.

Exemplu:

```
#define NULL_TELEM -1
```

În acest caz, directiva *define* s-a utilizat pentru a defini constanta simbolică NULL_TELEM ca având valoarea -1. Astfel, anterior compilării efective, toate aparițiile constantei simbolice NULL_TELEM vor fi înlocuite cu valoarea -1.

- Operatorii **->** și **.**

- Permit referirea datelor-membru și a metodelor claselor. Operatorul **.** este aplicat obiectului în sine, iar operatorul -> este folosit împreună cu un pointer la un obiect.

Exemple:

```
1.  Colectie c;
2.  assert(c.dim() == 0);
3.  assert(c.vida() == true);
```

În exemplul anterior, se folosește operatorul **.** pentru referirea metodelor clasei *Colectie*. Operatorul este aplicat obiectului *c* de tip *Colectie.*

```
1.  int Colectie::dim() const {
2.      return this->len;
3.  }
```

În exemplul anterior, se folosește operatorul **->** pentru referirea variabilei-membru *len* a clasei *Colectie.* Operatorul este aplicat pointerului *this* care referă obiectul curent, în interiorul clasei.

- **Alocarea unui tablou (*array*) unidimensional**
  - **Alocarea statică**

Sintaxa pentru **definirea unui tablou unidimensional prin alocare statică** în C++ este următoarea:

tip_elemente nume_tablou [număr_elemente]

*număr_elemente* trebuie să fie o expresie constantă întrucât, în acest caz, discutăm despre blocuri de memorie statică a căror dimensiune trebuie să fie determinată în momentul compilării, deci anterior execuției programului.

OBS: Spațiul de memorie ocupat de un tablou static este dealocat automat, în momentul în care tabloul nu mai este vizibil (acesta devine *out of scope*).

  - **Alocarea dinamică**

Pentru alocarea dinamică a unui tablou unidimensional în C++ se pot folosi operatorii ***new*** și ***delete***. Sintaxa pentru aceasta este următoarea:

tip_elemente* nume_pointer_tablou = **new** tip_elemente [număr_elemente]

Prin aceasta, se va aloca, în manieră dinamică, spațiu pentru *număr_elemente* elemente de tipul *tip_elemente* și returnează un pointer, *nume_pointer_tablou*, la primul element din secvență. Primul element poate fi accesat fie prin expresia nume_pointer_tablou[0], fie prin expresia * nume_pointer_tablou.

Dacă în cazul unui tablou unidimensional definit prin alocare statică, *număr_elemente* trebuie să fie o expresie constantă și, astfel, dimensiunea lui trebuie să determinată în momentul compilării, alocarea dinamică a memoriei efectuată prin utilizarea operatorului *new* permite stabilirea dimensiunii tabloului unidimensional în timpul execuției, folosind o valoare variabilă pentru *număr_elemente.*

OBS: Spațiul de memorie ocupat de un tablou alocat dinamic **nu** este dealocat automat, dealocarea revenind în responsabilitatea programatorului. Astfel, în momentul în care memoria alocată dinamic nu mai este necesară scopului pentru care a fost alocată, ea va fi eliberată pentru a putea fi reutilizată pentru alte alocări dinamice. În acest scop, se va utiliza operatorul ***delete***, conform următoarei sintaxe:

**delete**[] nume_pointer_tablou

Exemplu:

```
1.  //Colectie.h
2.
3.  class Colectie {
4.
5.      friend class IteratorColectie;
6.
7.  private:
8.      int len;
9.      int cap;
10.     int* elemente;
11.
12. //Colectie.cpp
13.
14. Colectie::Colectie() {
15.     this->cap = 10;
16.     this->len = 0;
17.     this->elemente = new int[this->cap];
18. }
19.
20. Colectie::~Colectie() {
21.     if(this->elemente!=NULL)
22.         delete[] this->elemente;
23. }
```

În exemplul anterior, tabloul unidimensional *elemente* se alocă dinamic, în cadrul constructorului clasei *Colectie,* dimensiunea lui fiind stabilită în momentul execuției și dată de variabila-membru a clasei *cap.* În cadrul destructorului, spațiul de memorie ocupat de tablou este dealocat prin folosirea operatorului *delete*.

Alternativ, pentru alocarea dinamică a unui tablou unidimensional în C++ se pot folosi și funcțiile ***malloc / calloc*** și ***free,*** definite în *header*-ul <cstdlib> (cunoscut ca <stdlib.h> in C). Sintaxa pentru aceasta este următoarea:

tip_elemente* nume_pointer_tablou = (tip_elemente*) **malloc** (număr_elemente*sizeof(tip_elemente))

OBS: *malloc* alocă memorie fără să o inițializeze. Spre deosebire de aceasta, *calloc* alocă memorie și inițializează toți biții cu 0.

Pentru dealocare, care revine, și în acest caz, în responsabilitatea programatorului, se va folosi sintaxa:

**free**(nume_pointer_tablou)

- **Constructor**

Un constructor în C++ este o metodă specială care se apelează în mod automat la crearea unui obiect al unei clase. Pentru definirea unui constructor, se va folosi **același nume precum cel al clasei**.

Nu are tip returnat.

Rolul constructorului este de a inițializa atributele (datele-membru).

Spre deosebire de funcțiile obișnuite, constructorii pot prezenta o **listă de inițializare** pentru atribute de forma:

nume_clasă(parametri_constructor) : atribut_1(paramentru1), ..., atribut_n(parametru_n) {...}

Exemple:

```
1.  //Colectie.h
2.
3.  public:
4.      //constructorul implicit
5.      Colectie();
6.
7.  //Colectie.cpp
8.
9.  Colectie::Colectie() {
10.     this->cap = 10;
11.     this->len = 0;
12.     this->elemente = new int[this->cap];
13. }
```

```
1.  //IteratorColectie.h
2.
3.  private:
4.      const Colectie& col;
5.      int pozCurenta;
6.      IteratorColectie(const Colectie& c);
7.
8.  //IteratorColectie.cpp
9.
10. IteratorColectie::IteratorColectie(const Colectie& c): col(c) {
11.     this->pozCurenta = 0;
12. }
```

În cel de-al doilea exemplu, constructorul prezintă o listă de inițializare în cadrul căreia se inițializează referința *col* la colecția iterată.

- **Destructor ~**

Destructorii sunt metode speciale apelate de către compilator la ştergerea din memorie a obiectelor.

Numele destructorilor este format din caracterul ~ şi numele clasei.
Destructorii nu au tip returnat şi nu pot primi parametri.

Exemplu:

```
1.  Colectie::~Colectie() {
2.      if(this->elemente!=NULL)
3.          delete[] this->elemente;
4.  }
```

- **Excepții**

Apariția excepțiilor (generate de încălcarea precondițiilor) se semnalează prin intermediul cuvântului-cheie ***throw,*** conform următoarei sintaxe:

**throw** tip_excepție;

Exemplu:

throw exception();

Clasa *exception* este definită în *header*-ul <exception>, în spațiul de nume *std* (în care este împachetată librăria standard C++), aceasta fiind clasa de bază pentru toate excepțiile standard.

Pentru semnalarea unei excepții de tipul *exception*, deci, este necesară includerea *header*-ului <exception>, folosind sintaxa:

#include exception;

Pentru a evita prefixarea cu *std::*, se poate utiliza directiva *using,* care expune toate elementele unui spatiu de nume in unitatea din care a fost apelata.

using namespace std;

Excepțiile se captează prin blocuri ***catch***, acestea urmând unui block ***try.***

```
try  {
        // cod
        throw tip_excepție;
}
catch(TipExceptie){
         // cod tratare excepție
}
```

Exemple:

```
1.  void IteratorLP::urmator(){
2.      if (!valid()){
3.          throw std::exception();
4.      }
5.      this->pozCurenta++;
6.  }
```

În exemplul anterior, s-a ridicat o excepție de tipul de bază *exception*. Nefiind folosită directiva *using* pentru expunerea tuturor elementelor spațiului de nume *std*, s-a prefixat constructorul tipul *exception* cu denumirea spațiului de nume (std::exception()).

```
1.  void testAllExtins() {
2.      Lista lista = Lista(); // creem lista vida
3.      assert(lista.vida());
4.      IteratorLP it0 = lista.prim(); //creem un iterator pe lista vida
5.      assert(!it0.valid());
6.      try {
```

```
            it0.element();  // nu ar trebui sa fie posibil ca iteratorul sa returneze un element
            assert(false);
        } catch (exception&) {
            assert(true);
        }
        //[...]
}
```

În exemplul anterior, s-au folosit blocurile *try* și *catch* pentru captarea excepțiilor.

- **const**

- Cuvânt-cheie care permite declararea unui pointer sau a unei referințe la o valoare constantă.

```
class IteratorColectie
{
    friend class Colectie;
private:
    IteratorColectie(const Colectie& c);
    const Colectie& col;
    //[...]
```

În exemplul anterior, referințele la un obiect de tipul *Colecție c* și *col* sunt constante (astfel nepermițându-se modificarea obiectului de tip *Colectie*).

- **Pointeri la funcție**

Exemple:

A.

```
//Colectie.h

typedef bool(*Relatie)(TElem, TElem);
bool rel(TElem, TElem);

//Colecție.cpp

bool rel(TElem e1, TElem e2) {
    if (e1 <= e2) return true;
        else return false;
}
```

```
//Colecție.cpp

bool Colectie::cauta(TElem elem) const {
    int index = 0;
    while (index < this->len && rel(elem, this->elemente[index]) == false) {
        index++;
    }
    if (index == this->len || this->elemente[index] != elem) {
        return false;
    }
```

```
    else {
        return true;
    }
}
```

B.

```
//LO.h

typedef int TComparabil;
typedef TComparabil TElement;

typedef bool (*Relatie)(TElement, TElement);


class LO {

private:
    //[...]
    Relatie relatie;
```

```
//LO.cpp

LO::LO(Relatie r) {
    //[...]
    this->relatie = r;
}

int LO::cauta(TElement e) const {
    int pozitie = 0;
    while (pozitie < this->lungime && this->relatie(this->elemente[pozitie], e)
            && !this->relatie(e, this->elemente[pozitie])) {
        pozitie++;
    }
    if (pozitie == this->lungime
            || !(this->relatie(this->elemente[pozitie], e)
                    && this->relatie(e, this->elemente[pozitie]))) {
        return -1;
    } else {
        return pozitie;
    }
}
```

```
//TestExtins.cpp

//Relatie de ordine - crescator
bool cresc(TElement c1, TElement c2) {
    if (c1 <= c2) {
        return true;
    } else {
        return false;
```

```
9.          }
10.     }
11.
12.
13.     void testCreeaza() {
14.         LO lo = LO(cresc);
15.         //[...]
16.     }
```

În ambele exemple *Relatie* este declarat de tipul pointer la funcție cu doi parametri de tip *TElement* și rezultat de tipul *bool* .

În cazul primului exemplu, relația nu este expusă în interfață (nefiind transmisă ca parametru pentru constructor), pe când în cazul celui de-al doilea exemplu aceasta este expusă în interfață, constructorul fiind parametrizat o relație a.î. să se construiască o listă ordonată a căror elemente vor respecta ordinea impusă de relația transmisă ca parametru.

Observăm că *rel* (în cazul primului exemplu) și *cresc (*în cazul celui de-al doilea exemplu) respectă signatura indicată.

- **bool**

  Tipul de date **bool** include valorile **true (1)** și **false (0)**.

- **Operatori logici: &&, ||, !**

Operatorii logici au operanzi de tip valori de adevăr și, de asemenea, au ca rezultat o valoare de adevăr.

În C++, operatorii logici pot fi aplicați inclusiv valorilor numerice și au ca rezultat una din valorile 0 sau 1. În exemplele de mai jos vom folosi literalii true și false, de tip *bool*.

**Negația: !**

- ! true este false. Orice valoare nenulă negată devine 0.
- ! false este true. 0 negat devine 1.

**Disjuncția: ||**

- false || false → false
- false || true → true
- true || false → true
- true || true → true

**Conjuncția: &&**

- false && false → false
- false && true → false
- true && false → false
- true && true → true