LABORATORUL 3

1 assert

`assert` este o funcție din headerul `cassert` cu următoarea signatură `void assert(int)` și funcționează în următorul fel: dacă parametrul primit este evaluat la 0 atunci programul este întrerupt folosing funcția `abort` , altfel se continuă execuția programului. Exemplu:

```
#include <iostream>
#include <cassert>

int foo (int a, int b = 1) {
    if (a % 2 == 0) {
        return a + b;
    }
    return a - b;
}

int main () {
    assert(foo(2) == 3);
    assert(foo(2, 4) == 6);
    assert(foo(5, 44) == -39);
    assert(foo(17) == 16 );
    // assert(foo(3, 3) == 4);  va face ca programul sa fie intrerupt
    std::cout << "Functia foo testa cu succes" << std::endl;
    return 0;
}
```

Funcția `assert` ne ajută să scriem teste funcționalitățile în codul nostru. De ce este important să scriem teste? În primul rând ne asigurăm că toate funcținalitățile sunt implementate corect. În al doilea rând, putem valida oricând printr-o simpla rulare a testelor dacă codul nostru încă funcționează după ce am făcut modificări pe el.

2 Excepții

Excepțiile oferă posibilitatea de a trata situații "speciale" la momentul rulării (erori la runtime) prin transferul execuției către o zonă de cod care va gestiona eroarea. Pentru a putea gestiona o execuție trebuie ca setul de instrucțiuni care ar putea genera excepția să fie inclus într-un bloc `try-catch` . Exemplu:

```
#include <iostream>
using namespace std;
int main () {
    try {
        int i;
        cin >> i;
```

```
        if (i % 2) {
            throw i;                    // arunca i ca exceptie daca este impar
        }
        cout << i << " este par";
    } catch (int x) {                   // gestioneaza exceptia
        cout << x << " este impar";
    }
    cout << endl;
    return 0;
}
```

Putem avea mai multe zone `catch` înlănțuite, fiecare oferind gestiunea unui tip diferit de excepție. Dacă nu stim ce tip de excepție se aruncă putem folosi `...` pentru a oferi o gestiune generică.

```
try {

}
catch (int i) {}        // gestioneaza intregi
catch (string s) {}     // gestioneaza stringuri
...
catch (...) {}          // gestioneaza tot ce nu e tratat mai sus
```

În C++ avem definită clasa abstractă `exception` (prezentă în headerul cu același nume) utilizată în gestiunea excepțiilor. Aceasta expune metoda `what` care întoarce `const char*` (o descriere a excepției). Este recomandat ca atunci când în code trebuie definite situații excepționale se recomandă aruncarea de excepții care moștenesc clasa `exception` (definite de utilizator sau nu). Tipuri de excepții deja definite:

- `bad_alloc`
- `bad_cast`
- `bad_exception`
- `bad_function_call`
- `bad_typeid`
- `bad_weak_ptr`
- `ios_base::failure`
- `logic_error`
- `runtime_error`
- `domain_error`
- `future_error`
- `invalid_argument`
- `length_error`
- `out_of_range`
- `overflow_error`
- `range_error`
- `system_error`
- `underflow_error`
- `bad_array_new_length`

Putem specifica dacă o funcție/metodă aruncă excepții folosind specificatorul `noexcept`. De asemenea putem verifica dacă o expresie aruncă o excepție sau nu folosind operatorul cu același nume: `noexcept(<expr>)` întoarce `true` dacă `<expr>` nu aruncă o excepție, `false` dacă e posibil ca `<expr>` să arunce o excepție.

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

void f () noexcept;
void g ();

int main () {
    cout << "f nu arunca exceptii?" << noexcept(f()) << endl;
    cout << "g nu arunca exceptii?" << noexcept(g()) << endl;
```

```
    return 0;
}
// f nu arunca exceptii? 1
// g nu arunca exceptii? 0
```

3 Listă de inițializare

Atunci când implementăm un constructor, imediat după lista de parametri, putem enumera felul în care se inițializează câmpurile clasei. În lista de inițializare putem specifica cum să apelăm constructorul pentru câmpurile alocate static, inițializa câmpurile const sau specifica apeluri explicite ale constructorilor clasei de bază (la moștenire). Lista de inițializare poate fi folosită atât la declarații inline implicite cât și în cazul declarărilor normale. Sintaxa:

```
class <nume_clasa> {
    /*
        definitii campuri
    */
    public:
        /*
            definitii metode
        */
        <nume_clasa> (<lista_parametri>);
};


<nume_clasa>::<nume_clasa>(<lista_parametri>) : <lista_initializare> {
    // implementare constructor
}
```

Exemplu:

```
    class A {
        int x;
        string s;
        double *d;
        public:
           A () : x(1), s("202345"), d(NULL) {}
    };

```

4 Metode și câmpuri statice

Câmpurile statice sunt proprietăți care sunt împărțite de toate obiectele unei clase. Un câmp static este inițializat cu valoarea default pentru tipul de date declarat, dacă altă inițializare nu este specificată. Pentru a declara un câmp static trebuie folosit cuvântul cheie *static*. Sintaxă:

```
class <nume_clasa> {
    /*
        definitii campuri
    */
    static <tip_date> <nume_camp>;
    public:
    /*
        definitii metode
    */
```

```
};

// initializare camp static
<tip_date> <nume_clasa>::<nume_camp> = <valoare_initiala>;
```

O clasă poate avea și *metode statice*. Metodele statice nu au pointerul **this** și pot accesa doar câmpuri statice și pot apela doar alte metode statice ale clasei. Metodele statice nu pot fi apelate folosind obiect, singura modalitate de apelare fiind folosirea operatorului rezoluție de scop:

```
    <nume_clasa>::<nume_metoda>(parametri);
```

Pentru a declara o metodă statică e sufcient să adăugă cuvântul cheie static în fața signaturii metodei. Sintaxă:

```
class <nume_clasa> {
    /*
        definitii campuri
    */
    public:
    /*
        definitii metode
    */
    static <tip_retur> <nume_metoda>(lista_parametri);
};

<tip_retur> <nume_clasa>::<nume_metoda>(lista_parametri) {
    /* implementare */
}
```

Exercitiu

Implementați o aplicație care gestioneaza înscrierea studenților pentru examenul de admitere. Pentru fiecare student în aplicație se vor introduce:

- nume (`string`)
- prenume (`string`)
- medie examen bacalaureat (`float`)
- specializarea/specializarile la care se face înscrierea `string[]`

După inscriere, fiecare candidat primește un cod unic de identificare de forma FMI_<nr_id> (unde nr_id este numarul de ordine al candidatului – al câtelea candidat înscris). Aplicația trebuie sa ofere urmatoare funționalități:

- adăugarea unui nou canditat;
- afișarea candidaților înscriși la o anumită specialiare;
- eliminarea unui candidat folosind codul de de identificare;

Folosiți cât mai multe elemente de POO învățate! Memoria va fi alocată dinamic.