Tutoriat 2

Keyword ”const”

Sunt anumite situaţii în proiectarea unei clase în care este necesar ca anumite date să nu poată să fie modificate. Pentru a realiza acest lucru, avem nevoie de un cuvânt cheie care la prima vedere e inofensiv, dar poate provoca multe bug-uri greu de depistat şi, de aceea, trebuie folosit cu imensă atenţie. Acest cuvânt este **const.**

Exemple:

- pentru clasa *Student*, dacă vrem să adăugăm un câmp *dataNasterii*, logic ar fi ca acest câmp să fie constant, pentru că nu se va schimba data naşterii unei persoane;
- pentru o clasă *CardBancar,* un câmp de date *numarCard* trebuie păstrat constant;

Exemple de declarare ale unor variabile constante:

```
1 const int x = 3;
2 const double y = 3.0;
3 // .........
```

Acum că am observat cum se declară nişte variabile obişnuite constante, trebuie să discutăm despre ce face de fapt acest cuvânt cheie ***const.*** Acesta este un *flag (semnal)* către compilator, care îi spune să nu permită vreodată modificarea acelei variabile. Orice încercare de modificare produce automat o *eroare de compilare.*

Obs. Dacă nu avem voie să modificăm o variabilă declarată folosind ***const***, atunci putem deduce următorul lucru: pentru a putea lucra cu o astfel de variabilă, aceasta **TREBUIE INIŢIALIZATĂ.** Dacă omitem acest pas, nu primim o eroare de compilare, dar vom întâmpina probleme la runtime, pentru că acea variabilă are o valoare default dată de compilator şi, cu siguranţă, nu ne dorim asta.

Pointeri constanţi vs pointeri către constante

Când declarăm variabile constante obişnuite nu avem probleme cu ordinea declarării. Adică:

```
1 const int x = 3;    <==>    int const x = 3;
```

Problemele apar când dorim declararea unor pointeri şi avem nevoie şi de const. Aici sunt 2 tipuri de declarări:

```
1 const int* p;    !=    int* const p;
```

Pentru a rezolva această problemă ne folosim de citirea tipului de date de la dreapta la stânga, astfel:

**const int *** p = pointer către un întreg constant (un pointer către o valoare care nu se poate modifica, dar pointerul se poate muta către altă zonă de memorie);

p = &x; // unde x e un nr întreg. **E permisă această operaţie**.
*p = 5; // operaţia este interzisă. Nu avem voie să modificăm valoarea lui p.

int * const p = pointer constant către un întreg (un pointer către o valoare întreagă care se poate modifica, dar pointerul nu se poate muta).

p = &x; // operaţia este interzisă. Nu avem voie să mutăm pointerul.
*p = 5; // **E permisă această operaţie**.

Date membre constante

Bineînţeles că şi clasele pot avea date constante. Acestea se comportă la fel ca variabilele constante. Numai că apare o problemă. Cum iniţializăm aceste date?
Răspuns: Lista de iniţializare din constructor.
Întrebare: Ce e aia?

Lista de iniţializare

Este o modalitate prin care datele unei clase pot fi iniţializate în constructor. Se apelează înaintea instrucţiunilor din interiorul constructorului. Este crucială pentru datele constante. Hai să vedem cum arată o listă de iniţializare:

```
class Student {
    const string CNP;
    int varsta;

  public:
    /* Constructor cu lista de init */
    Student(string _CNP) : CNP(_CNP) {
        varsta = 0;
    }

    /* Constructor fara lista */
    Student(int _varsta) : CNP("") {
        varsta = _varsta;
    }
};
```

Putem observa că lista de inițializare este marcată prin caracterul ”**:**”. După acest caracter, urmează datele clasei, iar între paranteze valoarea cu care inițializăm aceste date. Adică, *CNP* va primi valoarea parametrului *_CNP.*

Această listă poate fi folosită pentru orice tip de variabile, dar este absolut necesară pentru datele constante. Dacă la primul constructor am fi făcut inițializarea între acolade, ar fi fost generată o eroare de compilare, fiindcă nu putem aplica operatorul = pentru o variabilă constantă.

Metode constante

Acum vom discuta de una dintre cele mai iritante probleme generate de cuvântul cheie *const.* După cum le spune numele, metodele constante nu au voie să schimbe valoarea vreunul câmp al clasei apelante. Acestea sunt specifice unei clase, deci nu pot fi declarate altundeva decât în interiorul unei clase. *Exemplu:*

```
class Student {
    const string CNP;
    int varsta;

  public:
    int getVarsta() const {
       return varsta;
    }
};
```

Acel cuvânt cheie îi spune compilatorului că ce se întâmplă în interiorul metodei, nu are voie să ducă în vreun fel la modificarea unei date a clasei, mai exact a pointer-ului *this.* Mai exact, următoarea secvenţă ar fi fost **GREŞITĂ**.

```
1 void met() const {
2    varsta = 30;
3 }
```

Această bucată de cod ar fi generat o eroare de compilare, fiindcă se încearcă modificarea datei *varsta.*

În general, metodele de tip ***getter*** sunt declarate constante. De foarte multe ori, lipsa acestui cuvânt cheie poate cauza erori greu de depistat.

Erori comune provocate de const

- ***Obiecte constante***

Ca orice variabile, şi obiectele pot fi declarate constante. Acest lucru îl obligă pe cel care implementează clasa să garanteze că acel obiect nu va fi modificat. Asta se face bineînţeles prin **METODE CONSTANTE**.

Un obiect constant poate apela doar metode constante.

Asta cauzează multă confuzie. Vom vedea de ce în exemplul următor:

```
1 class Student {
2    int varsta;
3
4   public:
5     int getVarsta() {
6        return varsta;
7     }
8 };
9
10 int main() {
11    const Student s;
12    cout << s.getVarsta(); // eroare
13 }
```

Obiectul ***constant*** s încearcă să apeleze o metodă ***neconstantă.*** Chiar dacă metoda *getVarsta()* nu face nicio modificare asupra obiectului, tot este incorect,

pentru că nu a fost specificat cuvântul *const.* Varianta corectă este următoarea:

```
1 class Student {
2    int varsta;
3
4   public:
5     int getVarsta() const {
6        return varsta;
7     }
8 };
9
10 int main() {
11    const Student s;
12    cout << s.getVarsta();
13 }
```

Acel simplu *const* adăugat la ***linia 5*** a fost diferența dintre un cod executat cu succes și unul care provoacă o eroare.

Referințe neconstante

Trebuie oferită atenție metodelor care primesc ca parametri obiecte transmise prin *referință*. Acea referință presupune că acele obiecte ***pot fi modificate.***

```
1 class Student {
2    int varsta;
3
4   public:
5     void sum(Student& s) {
6        cout << varsta + s.varsta;
7     }
8 };
9
10 int main() {
11    Student s1;
12    const Student s2;
13
14    cout << s1.sum(s2); // eroare
15 }
```

Acest cod provoacă o eroare, fiindcă pe ***linia 14*** încercăm să apelăm metoda *sum* care primește ca parametru o ***referință*** la un student. Problema apare din cauză că acea referință poate fi modificată în metodă, ceea ce încalcă promisiunea lui ***const.*** Compilatorul nu se ”uită” în metodă să vadă dacă chiar este modificat s. Doar vede că nu este garantat că va rămâne constant, așa că întoarce o eroare.

```
1 class Student {
2    int varsta;
3
4   public:
5     void sum(const Student& s) {
6        cout << varsta + s.varsta;
7     }
8 };
9
10 int main() {
11    Student s1;
12    const Student s2;
13
14    cout << s1.sum(s2);
15 }
```

Acest cod rulează, pentru că am adăugat la ***linia 5*** cuvântul ***const*** lângă referința către un Student. Acest lucru îi garantează compilatorului că obiectul nu se va modifica și permite execuția cu succes.

Acum, dacă în metoda *sum* am fi încercat modificarea câmpului *varsta,* e clar că ar fi cauzată o eroare.

Keyword ”static”

C

În C, exista noțiunea de variabile statice. Acestea puteau fi declarate în orice funcție și se comportau ca niște variabile globale. Sunt inițializate o singură dată la primul apel al funcției. *Exemplu:*

```
void f() {
   static int nr = 0;
   nr++;
   printf("%d", nr);
}

int main() {
   f(); // 1
   f(); // 2
}
```

În această bucată de cod, iniţializarea variabilei *nr* se face doar la primul apel al funcţiei f, apoi valoarea rămânând neschimbată, exact ca la o variabilă globală.

C++

Variabilele statice au rămas neschimbate în *C++.* Discuţia aici intervine aici despre ***membri statici*** ai unei *clase.*

Orice clasă poate să aibă *date* şi *metode* statice. Comportamentul este asemănător ca la *variabilele* statice din funcţii.

Membri statici ai unei clase nu aparţin de o instanţă, ci de întreaga clasă.

Ce înseamnă asta?

Un membru static este ceva *global* clasei. Adică, ***are aceeaşi valoare pentru orice instanţă a clasei.***

Date statice

Datele statice sunt primele iniţializate într-o clasă şi nu aparţin direct de clasă. Astfel, ele **NU** pot fi iniţializate în constructor. Există 2 tipuri de date statice care aduc 2 moduri diferite de iniţializare:

- ***neconstante*** = acestea sunt mai întalnite.

```
1 class A {
2   public:
3       static int x;
4 };
5
6 int A::x = 3;
7
8 int main() {
9     cout << A::x; // 3
10    A::x++;
11
12    A a;
13    cout << a.x; // 4
14 }
```

La ***linia 6*** putem vedea cum se iniţializează o variabilă statică. Această linie este **OBLIGATORIE** pentru funcţionarea programului.

Dacă omitem acea linie nu primim imediat o eroare de compilare, ci numai dacă încercăm în vreun moment să accesăm acea dată statică.

Obs. Putem accesa o dată statică atât cu *operatorul de rezoluţie(:)* precum şi cu o instanţă a clasei. *NU* e indicată a doua metode (cu o instanţă) pentru că nu are sens folosirea unei instanţe ţinând cont de definiţia unei date statice (nu aparţine de un obiect).

- ***constante***

```
1 class A {
2   public:
3       const static int x = 3;
4 };
5
6 int main() {
7     cout << A::x; // 3
8 }
```

Putem observa că iniţializarea se face la ***linia 3.*** Iniţializarea de data trecută nu funcţionează pentru datele statice constante, cum nici invers nu e cazul.

Metode statice

Acestea respectă principiul cuvântului cheie *static.* O metodă statică poate fi iniţializată fie în clasă fie în afara sa.

```
1 class A {
2   public:
3       static void m1() {
4           cout << "m1";
5       }
6       static void m2();
7 };
8
9 void A::m2() {
10    cout << "m2";
11 }
12
13 int main() {
14     A::m1(); // m1
15
16     A a;
17     a.m2(); // m2
18 }
```

Rămâne valabil ce am spus la date statice legat de modul de a accesa o metodă statică.

O metodă statică are acces doar la datele şi metodele STATICE ale unei clase.

Erori comune provocate de static

- ***Dată statică neiniţializată***

```
1 class A {
2   public:
3      static int x;
4 };
5
6 int main() {
7     cout << A::x; // eroare
8 }
```

Greşit

```
1 class A {
2   public:
3      static int x;
4 };
5
6 int A::x = 3;
7
8 int main() {
9     cout << A::x; // 3
10 }
```

Corect

Prima secvenţă provoacă o *eroare de compilare,* din cauză că se încearcă folosirea unei date statice *neiniţializate*.

Problema se rezolvă în a doua secvenţă unde s-a adăugat ***linia 6***.

- ***Membri nestatici în metode statice***

```
1 class A {
2   public:
3      int x = 3;
4      static void m() {
5         cout << x;
6      }
7 };
8
9 int main() {
10     A::m(); // eroare
11 }
```

După cum am spus, **NU** avem voie să accesăm nimic nestatic într-o metodă statică. Trebuie acordată mare atenţie acestor erori, pentru că sunt uşor de trecut cu vederea.

Putem rezolva eroarea de mai sus în mai multe moduri:

- *îl facem pe x dată statică;*
- *o facem pe m() o metodă nestatică;*
- *eliminăm pur şi simplu linia 5;*
- ***RECOMMENDED:** declarăm un obiect în interiorul lui m() şi accesăm prin instanţă:*

```
1 class A {
2   public:
3       int x = 3;
4       static void m() {
5           A a;
6           cout << a.x;
7       }
8 };
9
10 int main() {
11     A::m(); // 3
12 }
```