Exerciții cu modificatorul `const`

Motivație

Toate variabilele pe care le declarăm într-un program C++ sunt implicit **mutabile**, adică le putem modifica după ce le-am inițializat (i.e. putem schimba valoarea reținută în acea zonă de memorie). Din păcate, foarte multe bug-uri în practică apar din cauză că modificăm neintenționat anumite date.

De exemplu, poate că găsiți indicele unui element într-un vector, apelați un subprogram (care primește vectorul ca parametru) și apoi faceți ceva cu elementul de pe poziția i. Ce se întâmplă dacă subprogramul respectiv inserează un nou element în vector înainte de cel de pe poziția i? Indicele pe care l-ați găsit nu va mai fi valid. Limbajul oferă o măsură de prevenție pentru astfel de situații, prin cuvântul cheie/modificatorul `const`.

`const` este o promisiune pe care i-o facem compilatorului; îi indicăm că nu intenționăm să modificăm o variabilă după ce am declarat-o și inițializat-o, iar el se va asigura de acest lucru.

Referințe utile: *Write safer and cleaner code by leveraging the power of "Immutability"*, *Mutability & Immutability*.

Variabile `const`

Toate aceste exerciții se pot rezolva în subprogramul principal.

1. Definiți o variabilă locală de tip `const int`, inițializată cu un număr întreg ales de voi.

   Încercați să atribuiți o nouă valoare variabilei (după inițializare). Ce mesaj de eroare primiți? Când este detectată eroarea (la compilare sau în momentul execuției)?

2. Declarați o variabilă de tip `int&`. Puteți să o inițializați cu o referință la o variabilă de tip `const int`? De ce credeți că previne acest lucru compilatorul?

3. Declarați o variabilă locală de tip `const int&`. Încercați să o inițializați cu o referință la o variabilă de tip `const int`.

   Declarați o altă variabilă locală de tip `const int&` și încercați să o inițializați folosind o variabilă de tip `int`.

   De ce funcționează ambele metode?

4. Declarați o variabilă de tip `const int&` și inițializați-o cu constanta `42` (sau cu ce număr întreg vreți voi). Toate valorile literale/constante din cod au implicit tipul de date `const int` / `const char[]` / etc.

   Același lucru se aplică și pentru valorile temporare/„auxiliare". De exemplu, dacă am o funcție care primește un parametru de tipul `const int&`, o pot apela și cu expresia `2 * (3 + 5)`, dar nu pot face acest lucru dacă așteaptă un parametru de tip `int&`.

Pointeri `const`

Când lucrăm cu pointeri, trebuie să avem grijă deoarece putem pune modificatorul `const` în două locuri diferite.

Toate aceste exerciții se pot rezolva în subprogramul principal.

1. Creați o variabilă de tip `int*`. Inițializați-o cu adresa unei variabile de tip `int`.

   Modificați pointerul ca să trimită către o altă variabilă de tip `int`. În comparație cu referințele, pointerii pot să rețină adresa unor obiecte diferite pe parcursul vieții lor.

2. Declarați o variabilă de tip `int*`. Puteți să o faceți să rețină adresa unei variabile de tip `const int`? De ce credeți că previne acest lucru compilatorul?

   Observație: în funcție de compilatorul folosit și de setările acestuia, s-ar putea să nu primiți o eroare când încercați să rezolvați acest exercițiu, ci doar un *warning*.

3. Creați o variabilă de tip `int* const` și inițializați-o cu adresa unei variabile de tip `int`. Puteți să modificați valoarea variabilei respective prin intermediul acestui pointer? Puteți face acest pointer să trimită către un alt `int`?

   În momentul în care modificatorul `const` apare după `*`, obținem un pointer *constant* la o variabilă (posibil) *neconstantă*.

4. Creați o variabilă de tip `const int*` și inițializați-o cu adresa unei variabile de tip `int`. Puteți să modificați valoarea variabilei respective prin intermediul acestui pointer? Puteți face acest pointer să trimită către un alt `int`?

   În momentul în care modificatorul `const` apare înaintea `*`, obținem un pointer (posibil) *neconstant* la o variabilă *constantă*.

5. Creați o variabilă de tip `const int* const` și inițializați-o cu adresa unei variabile de tip `int`. Puteți să modificați valoarea variabilei respective prin

intermediul acestui pointer? Puteți face acest pointer să trimită către un alt `int`?

Metode `const`

Toate metodele (nestatice) unei clase au acces la obiectul implicit; pot accesa direct datele membru și metodele clasei, sau putem face acest lucru prin intermediul pointerului `this`.

Dacă nu declarăm altfel, obiectul implicit va fi (ca toate celelalte variabile din C++) *mutabil*. Cu alte cuvinte, dacă clasa noastră se numește `MyClass`, atunci `this` va avea tipul de date `MyClass* const` (pointerul nu poate fi schimbat, dar obiectul către care trimite **nu** este constant).

Ca să indicăm că o metodă din clasa noastră nu va modifica obiectul implicit, deci poate fi apelată și pe obiecte care sunt constante, trebuie să adăugăm modificatorul `const` după lista de parametrii formali:

```
1 int my_method() const
2 {
3     // nu pot modifica datele membru in aceasta metoda
4     // ...
5 }
```

1. Definiți o clasă `Date` care să rețină o dată calendaristică (ziua, luna și anul, reținute ca numere întregi). Adăugați un constructor pentru această clasă și metode getter pentru fiecare dintre variabilele membru.

   În subprogramul principal, creați un obiect de tip `const Data`. Modificați getteri să fie metode `const`, ca să îi puteți apela și pe acest obiect.

2. Definiți o clasă `DateInterval` care să rețină două obiecte de tipul `Date` (prima va fi anterioară cronologic față de a doua). Adăugați un constructor la această clasă care să inițializeze cele două date membru.

   Definiți o metodă `const` care să determine numărul de ani dintre cele două date stocate în clasă (ca număr întreg).

   Definiți o metodă cu altă metodă `const` care să determine numărul de decenii dintre cele două date stocate în clasă (ca număr întreg). Aceasta se va folosi de metoda definită anterior. Observați că în corpul acestei metode, nu puteți apela decât alte metode `const`, iar toate datele membru se comportă de parcă ar fi `const`.