

Projet C++ : Classe Serie

Objectif

Le but de ce projet est de créer une classe générique `Serie<T>` en C++, représentant une série de données similaire à un vecteur pandas en Python ou une colonne de DataFrame.

Cette classe ne devra pas utiliser de librairie extérieure (hormis la STL) et être codée entièrement par vous.

Attributs

Cette classe devra avoir les attributs suivants, accessibles par getters/setters :

- **data** : un conteneur stockant les éléments de type T (par exemple `std::vector<T>`)
- **name** : un nom optionnel pour la série (chaîne de caractères)

Constructeurs

La classe devra avoir les constructeurs suivants :

- Un constructeur par défaut, créant une série vide
- Un constructeur prenant un `std::vector<T>` (par copie)
- Un constructeur prenant une taille et une valeur de remplissage

Méthodes d'accès

- **operator[]** : accès par indice (versions const et non-const)
- **at()** : accès par indice avec vérification des bornes et exception si hors limites
- **size()** : retourne le nombre d'éléments
- **empty()** : vérifie si la série est vide
- **push_back()** : ajoute un élément à la fin
- **clear()** : vide la série
- **begin() / end()** : itérateurs pour parcourir la série

Opérateurs de comparaison (retournant Serie<bool>)

Les opérations de comparaison devront fonctionner **élément par élément** et retourner une **Serie<bool>** :

- **Comparaison avec une autre série** (via `==`, `!=`, `<`, `<=`, `>`, `>=`) :
 - Compare les éléments correspondants
 - La taille du résultat sera le minimum des deux tailles
- **Comparaison avec une valeur scalaire** (via `==`, `!=`, `<`, `<=`, `>`, `>=`) :
 - Compare chaque élément avec la valeur
 - La taille du résultat est identique à celle de la série

Exemples d'utilisation :

```
Serie<int> s1({1, 2, 3, 4, 5});  
Serie<int> s2({5, 4, 3, 2, 1});  
  
Serie<bool> mask1 = s1 > 3;           // {false, false, false, true, true}  
Serie<bool> mask2 = s1 == s2;         // {false, false, true, false, false}  
Serie<bool> mask3 = s1 <= 3;          // {true, true, true, false, false}
```

Indexation avancée

La classe devra supporter plusieurs types d'indexation via **operator[]** :

- 1. **Indexation booléenne** : `Serie<bool>` comme masque
 - Retourne une nouvelle série contenant uniquement les éléments où le masque est `true`
- 2. **Indexation par indices** : `Serie<int>` contenant des positions
 - Retourne une nouvelle série avec les éléments aux positions spécifiées
- 3. **Indexation par prédictat** : fonction ou lambda retournant un booléen
 - Retourne une nouvelle série contenant les éléments satisfaisant le prédictat

Exemples d'utilisation :

```
Serie<int> s({1, 2, 3, 4, 5, 6, 7, 8, 9, 10});  
  
// Indexation booléenne  
Serie<int> filtered1 = s[s > 5];           // {6, 7, 8, 9, 10}  
  
// Indexation combinée avec opérateurs logiques  
Serie<int> filtered2 = s[s > 3 && s <= 7]; // {4, 5, 6, 7}
```

```

// Indexation par prédictat (lambda)
auto is_even = [](int x) { return x % 2 == 0; };
Serie<int> evens = s[is_even];           // {2, 4, 6, 8, 10}

// Indexation par indices
Serie<int> indices({0, 2, 4, 6});
Serie<int> selected = s[indices];        // {1, 3, 5, 7}

```

Opérateurs logiques (retournant Serie<bool>)

- **Opérateurs avec une autre série** (via `&&`, `||`) : opérations élément par élément
- **Opérateur de négation** (via `!`) : inverse tous les booléens

Exemples d'utilisation :

```

Serie<int> s({1, 2, 3, 4, 5});

Serie<int> result = s[(s > 2) && (s < 5)]; // {3, 4}
Serie<int> result2 = s[!(s == 3)];           // {1, 2, 4, 5}

```

Opérateurs arithmétiques (optionnel, bonus)

Si le type T le permet, implémenter :

- **Addition** (via `+`, `+=`) : avec une autre série ou une valeur
- **Soustraction** (via `-`, `-=`) : avec une autre série ou une valeur
- **Multiplication** (via `*`, `*=`) : avec une autre série ou une valeur
- **Division** (via `/`, `/=`) : avec une autre série ou une valeur

Opérateurs d'affichage

- **operator<<** : affiche la série sous la forme `[val1, val2, val3, ...]`

Gestion des erreurs

- Lancer une exception (`std::out_of_range`) lors d'un accès hors limites avec `at()`
- Gérer correctement les cas où les séries ont des tailles différentes lors des opérations

Fichier main.cpp

Vous joindrez à votre projet un fichier `main.cpp` séparé, démontrant **toutes** les fonctionnalités de votre classe, notamment :

1. Création de séries de différentes manières
2. Opérations de comparaison et génération de masques
3. Filtrage avec indexation booléenne
4. Filtrage avec prédictats/lambdas
5. Indexation par positions
6. Combinaison de plusieurs conditions avec `&&` et `||`
7. Affichage et manipulation de séries

Exemple de main.cpp :

```
int main() {
    // 1. Crédation de séries
    Serie<int> s1({1, 2, 3, 4, 5, 6, 7, 8, 9, 10});
    Serie<int> s2(5, 100); // 5 éléments valant 100

    std::cout << "s1: " << s1 << std::endl;
    std::cout << "s2: " << s2 << std::endl;

    // 2. Filtrage simple
    Serie<int> greater_than_5 = s1[s1 > 5];
    std::cout << "s1 > 5: " << greater_than_5 << std::endl;

    // 3. Filtrage combiné
    Serie<int> range = s1[(s1 > 3) && (s1 <= 7)];
    std::cout << "3 < s1 <= 7: " << range << std::endl;

    // 4. Filtrage avec prédictat
    auto is_even = [] (int x) { return x % 2 == 0; };
    Serie<int> evens = s1[is_even];
    std::cout << "Even numbers: " << evens << std::endl;

    // 5. Indexation par positions
    Serie<int> indices({0, 2, 4, 6, 8});
```

```

Serie<int> selected = s1[indices];
std::cout << "Selected by indices: " << selected << std::endl;

// 6. Opérations bitwise
Serie<int> shifted = s1 << 1;
std::cout << "s1 << 1: " << shifted << std::endl;

// 7. Négation logique
Serie<int> not_equal_5 = s1[!(s1 == 5)];
std::cout << "s1 != 5: " << not_equal_5 << std::endl;

return 0;
}

```

Notation

La notation se fera sur :

- **La réalisation des fonctions et opérations demandées** (50%)
- **La qualité du code** : règle des 5, public/private, utilisation des const et des références, gestion de la mémoire (30%)
- **La qualité du main.cpp** : exhaustivité des tests et clarté des démonstrations (20%)

Conseils

- Pensez à utiliser `std::vector<T>` comme conteneur sous-jacent
- Utilisez les templates correctement pour gérer différents types
- Faites attention aux conversions de types (notamment entre `Serie<T>` et `Serie<bool>`)
- Testez votre code avec différents types : `int`, `double`, `bool`, etc.
- Commentez votre code de manière appropriée
- Respectez les conventions de nommage C++

Bonne chance !