Surcharge d'opérateurs

Exercice 1 — Opérateur « () »

On définit l'interface :

```
class ValeursAdmises {
 public :
  virtual  bool operator()(char val) const = 0;
};
```

L'opérateur « () » qui prend un argument `val` de type `char`, permettra d'utiliser une notation fonctionnelle sur l'objet pour savoir si une valeur `val` répond à certains critères vis à vis de l'objet. Nous allons étudier deux exemples :

1. Définissez `Intervalle` une sous-classe de `valeursAdmises` dont les objets représentent un intervalle de caractères minuscules pris dans l'ordre alphabétique entre une valeur `char min` et une valeur `char max`, considérés inclus. Si les paramètres de construction sont inconsistants alors on prendra `'a'` pour `min` et `'z'` pour `max`.
   La « fonction » correspondant à l'opérateur redéfini vérifiera si le caractère donné en argument est ou pas dans l'intervalle modélisé.
2. *Application :* Ecrivez une méthode `void analyse (const Intervalle &i)` qui analyse le contenu de l'intervalle `i` comme si c'était une boite noire, et qui en affiche les bornes. Dans cet exemple on affichera `min : e max : m`

   ```
   int main() {
    Intervalle interV('e', 'm');
    analyse(interV);
   }
   ```

3. Faites de même pour une classe `TableauValeurs` dont les objets encapsulent un tableau (au sens `[]`) de caractères avec sa taille, et dont la « fonction » va vérifier si un caractère donné est ou pas dans le `TableauValeurs`.
   Exemple d'utilisation :

   ```
   char tab[] {'b', 'o', 'n', 'j', 'o', 'r'};
   TableauValeurs tabV { tab, 6 };
    if(tabV('u')) cout << "␣étrange␣..." ;
    else cout<< "␣oublié␣!!" ;
   ```

4. Écrivez une fonction : `void filtre(const string &s, const ValeursAdmises&)` qui n'affichera de la chaîne `s` que les caractères qui sont admis.
   Exemple d'utilisation :

   ```
   filtre("au␣revoir",interV); // affiche "ei"
   filtre("au␣revoir",tabV);   // affiche "ror"
   ```

Exercice 2 ([5 points] Exam 2022-23)

On rappelle comment fonctionne un itérateur sur la classe `vector` :

```
vector<int> v {1,2,3};
for(vector<int>::iterator i{v.begin()} ; i != v.end(); ++i)   cout << *i;

for(vector<int>::reverse_iterator i{v.rbegin()} ; i != v.rend(); ++i)  cout << *i;
```

Qui affichent respectivement :

```
123
321
```

Explication : la classe `vector<int>` définit, en interne, deux classes publiques `iterator` et `reverse_iterator`. Dans l'exemple, `v.begin()`,`v.end()` et `v.rbegin()`, `v.rend()` en sont respectivement des instances.

Nous avons aussi la possibilité d'écrire de manière concise :

```
for (int i: v) cout << i;
```

cette syntaxe du `for` avec les deux points est une sorte de macro qui est traduite vers la première des formes que nous avons données ci dessus. Le type `iterator` implicitement utilisé par la macro est trouvé dans la classe déclarée de `v` (ce qui permet de généraliser la macro à toutes collections). Les noms `begin()` et `end()` seront toujours ceux des méthodes invoquées pour obtenir les bornes utiles au parcours.

On souhaite introduire une forme syntaxique maison pour parcourir simplement notre vecteur dans l'autre sens, mais il n'y a pas de moyen de redéfinir `":"` on s'oriente donc vers une solution qui aurait cette forme :

```
for (int i:wrap{v}) { cout << i ;}
```

Dont l'affichage devra être `321`. Proposez une solution en ne vous occupant que du cas des vecteurs d'entiers (c.à d ne cherchez pas à faire de templates).

Indications :

— il vous faut écrire une nouvelle classe `wrap` et la munir de ce qu'attend la macro :

— soyez rassurés : la macro `for ( : )` se comportera avec `wrap` de la même façon qu'elle le fait pour les `vector` ou pour une autre une collection.

Exercice 3

1. Definir une classe `Vecteur` qui représente un vecteur de $\mathbb{R}^2$ avec ses constructeurs.
2. Surcharger les opérateurs `==` et `!=`, afin de pouvoir tester si deux vecteurs sont égaux ou différents.
3. Surcharger les opérateurs `+` et `-` afin de pouvoir faire la somme et la différence de deux vecteurs, et l'opérateur `*` afin de pouvoir réaliser le produit scalaire de deux vecteurs.
4. Surcharger l'opérateur `[]` afin de pouvoir accéder à une coordonnées du vecteur [1]. Tester en particulier : `Vecteur v; v[0] = 9;`

1. Il est probable que vous ayez besoin de définir deux versions de cet opérateur, l'un qui laisse l'objet `const` et l'autre non

5. Ajouter une méthode `double norm()` renvoyant la norme du vecteur.
6. Testez largement, en particulier les cas où il y a des conversions implicites, décidez ou pas de permettre des valeurs par défaut.

Exercice 4 On complète l'exercice 2. Il est indispensable à présent que vous compiliez avec l'option `-fno-elide-constructors`. **Assurez-vous en !**

1. Que se passe t'il si on écrit :

   ```
   vector v{1,2,3};
   wrap w1 {v};
   wrap w2 {w1};
   for (int i:w2) cout << i;
   ```

   Corrigez si nécessaire.
2. Pouvez vous écrire : `for (int i: wrap{{1,2,3}}){cout << i;}`
3. (hors programme) Testez : `for (int i :wrap{wrap{1,2,3}}){cout << i;}`
   Pour résoudre cette question, nous vous devons une explication : les objets temporaires ne sont pas toujours copiés, vous devriez constater que le constructeur de copie n'est pas invoqué, nous sommes dans le cas où c'est un move constructor qui est choisi. Nous l'avions évoqué lorsque nous avons présenté la rule of three, et ... il y a une rule of five. Ici vous aurez besoin d'ajouter le constructeur

   ```
   wrap (wrap &&w) : attribut{move(w.attribut)} {
     cout << "visibilité␣du␣move␣ctor";
   }
   ```