# TP 5 — Pointeurs intelligents

## 1. Jeu d'échecs (2/2)

Polymorphisme, Constructeur virtuel, std::unique_ptr, Fichiers.

1. Écrire une classe `echiquier` contenant un ensemble de pièces (on n'utilisera pas un tableau à 2 dimensions de pièces, puisque chaque pièce connaît déjà ses coordonnées, mais un conteneur simple de type liste ou vector). La classe devra être définie de façon à ce qu'il soit inutile de définir un destructeur et disposera (au moins) des méthodes suivantes :
   constructeur par défaut (échiquier vide) ;
   ajout d'une pièce ;
   `valeurdujoueur` prenant comme paramètre une couleur et retournant la somme des valeurs de toutes les pièces du joueur de cette couleur.
2. Définir une méthode `deplacer` prenant comme paramètre une position `source` et une position `destination` et déplaçant la pièce située à la position `source` (si elle existe) à la position `destination` (si le déplacement est possible). Si une pièce de l'autre couleur que celle située à la `source` était située à la position `destination`, celle-ci est supprimée de l'échiquier et détruite. Si par contre, la position `destination` contenait une pièce de la même couleur que la `source`, le déplacement est interdit. La méthode retournera `true` si le déplacement a été effectué, `false` sinon.
3. Faire en sorte que les instances d'`echiquier` puissent être copiées (constructeur par recopie, opérateur d'affectation).
4. Écrire une méthode `affichage` prenant comme paramètre un `std::ostream` et affichant l'échiquier sous la forme d'un tableau de 8x8 cases, chaque case étant représentée par 2 caractères tels que `CN` pour le cavalier noir ou `RB` pour le roi blanc.
5. Écrire une méthode `sauvegarde` prenant comme paramètre un `std::ofstream` et écrivant dans le flux fichier la totalité des pièces sous la forme retournée par `tostring`, une pièce par ligne. http://en.cppreference.com/w/cpp/io/basic_ofstream
6. Écrire une méthode `chargement` prenant comme paramètre un `std::ifstream` qui lit les pièces présentes dans le ficher (au format utilisé dans la méthode `sauvegarde`) et rajoute à l'échiquier les pièces correspondantes. http://en.cppreference.com/w/cpp/io/basic_ifstream vous aurez besoin de la méthode `eof` http://en.cppreference.com/w/cpp/io/basic_ios/eof pour détecter la fin du fichier, et la lecture se fera ligne par ligne avec la fonction `std::getline` http://en.cppreference.com/w/cpp/string/basic_string/getline .
7. Écrire une méthode `depart` dans `echiquier` initialisant l'échiquier comme ci-contre. Certaines pièces (tours et fous) n'ont pas été définies ici, vous ne les créerez donc pas sur l'échiquier. Écrire le programme principal gérant le déroulement de la partie : les blancs commencent, et chaque joueur doit, chacun à son tour, choisir une pièce de sa couleur en saisissant une position (sous la forme de deux entiers)[1] et la déplacer dans une position acceptable (sous la forme de deux entiers)[2]. La partie se termine quand un roi est pris, dans ce cas, le joueur ayant pris le roi de l'adversaire est vainqueur.

8. *(question optionnelle)* Le jeu tel qu'il est défini ainsi n'a pas beaucoup d'intérêt parce que certaines règles n'ont pas été prises en compte : entre autres, la dame est beaucoup trop puissante. En fait,

1 Si la position saisie ne correspond pas à une pièce du joueur, la programme re-demandera la position.

2 Si la position saisie ne correspond pas à un déplacement possible, un message du type « déplacement impossible » sera affiché, et le joueur devra à nouveau sélectionner pièce à déplacer (par sa position) et position destination.

selon les règles des échecs, la dame ne peut pas passer « par dessus » d'autres pièces : elle doit s'arrêter à la première case occupée par une pièce adverse (ou avant). Il faudrait donc modifier `deplacer` de `piece` pour prendre en compte cela, mais cela requiert de connaître l'`echiquier` depuis la pièce, ou faire ce test dans une méthode d'`echiquier`. Réfléchissez aux implications de ces deux choix, faites un choix, et écrivez le code pour gérer cette règle.

## 2. Comptes bancaires

Polymorphisme, Conception objet, std::shared_ptr.

*N'attendez pas la dernière question pour tester votre code. Seules les méthodes qui doivent effectivement faire des opérations d'entrées-sorties feront des opérations d'entrées-sorties. Donc, ne pas faire d'affichage dans les méthodes (sauf si ça peut vous aider en phase de mise au point, évidemment).*

1. Un compte bancaire peut avoir pour propriétaire :
   - une *personne physique* (les éléments à mémoriser sont : nom, adresse, date de naissance (sous la forme d'une chaîne)),
   - une *société* (nom, adresse, (pointeur sur) personne physique gérant la société),
   - un *couple* (deux *personnes physiques* qui doivent avoir même adresse, repérées par deux pointeurs).
   Afin de s'assurer que les pointeurs pointent toujours vers une instance qui ne sera pas détruite par ailleurs, on utilisera des `std::shared_ptr`. Définir une classe abstraite pour tous les propriétaires de compte définissant les méthodes suivantes : `nom` retournant une chaîne de caractère (formée de deux noms dans le cas d'un couple) ;
   `type` retournant le type de propriétaire ;
   `adresse` ;
   `etiquetteexpedition` retournant une chaîne formée du (des) nom(s) suivi de l'adresse ;
   un opérateur de sortie sur un flux utilisé pour l'affichage de toutes les informations concernant le propriétaire.
2. Nous définirons donc plusieurs classes pour représenter les différents types de propriétaires. Afin de simplifier le travail de la personne qui va utiliser ces classes, nous ne voulons pas qu'elle ait à connaître ces différentes classes mais seulement la **racine** de la hiérarchie définie ici. Par exemple, pour tester notre code, depuis le `main`, nous n'utiliserons que l**e nom de la classe racine** et ferons appel à une (des) méthode(s) de création des objets `fabriqueproprietaire`, qui, en fonction des paramètres reçus, instancie telle ou telle sous-classe de la racine, mais dont le type de retour ne fait apparaître que la classe racine (la méthode retournera un `std::shared_ptr`). Cette technique permet de fournir une hiérarchie de classes à un autre développeur sans que cet autre développeur ait besoin d'instancier (et donc connaître) explicitement toutes les classes de la hiérarchie : il va passer par une (des) méthode(s) simple(s) de fabrication des objets. Faire en sorte que cette méthode soit le **seul** moyen d'instancier des propriétaires (et sous-classes de propriétaire), et qu'il soit impossible de copier des propriétaires.
3. Chaque *compte* est rattaché à un propriétaire et a un numéro unique (qui doit être attribué automatiquement par le constructeur). Le montant figurant sur chaque compte doit évidemment être mémorisé. Définir des méthodes permettant de :
   `verser` des espèces sur un compte ;
   `retirer` des espèces ;
   `virer` un montant d'un compte vers un autre. Un virement effectué entre deux comptes ayant le même propriétaire est gratuit, alors qu'un virement effectué entre deux comptes de propriétaires différents est facturé 1 € (au compte débité).
4. Il existe différents types de comptes :
   - Un *compte courant* dispose d'un découvert autorisé qui peut être variable et négocié avec le banquier. Par défaut (à l'ouverture du compte), le découvert autorisé est de 0 €. Un retrait ou un

virement qui dépasse le découvert autorisé sera refusé.
- Un *LDD* doit contenir au minimum 15 € et au maximum 12000 €. Contrairement au compte courant, ce compte rapporte des intérêts qui sont de 0,75 % par an. Ce taux est amené à évoluer (mais restera toujours le même pour tous les LDD). Les intérêts sont ajoutés directement sur le compte. Prévoir une méthode `appliquerinterets` qui sera appelée une fois par an et qui rajoute au compte le montant des intérêts calculés en fonction du montant du compte au moment de l'appel[1]. À noter que l'application des intérêts peut entraîner un dépassement du plafond du compte, mais cela est autorisé.
- Un *LEP* doit avoir pour propriétaire une personne physique, doit contenir de 30 à 7700 €, et rapporte 1,25 % par an. Le taux ainsi que le calcul d'intérêts aura le même fonctionnement que pour un LDD.

5. Définir une classe `banque` regroupant un ensemble de propriétaires. Vous pouvez par exemple utiliser un `std::vector` comme conteneur. Les copies de `banque` seront interdites. On veut écrire la classe `banque` en respectant la règle des 0 et nous n'écrirons donc ni destructeur, ni constructeur par recopie, ni opérateur d'affectation, si ce n'est pour interdire les copies (mais, évidemment, la classe ne doit pas avoir de fuites de mémoire), et ferons en sorte d'écrire un code robuste. Avant de passer à la suite, vérifiez que votre classe `banque` fonctionne correctement et gère bien la mémoire (utilisez `valgrind` pour vérifier l'absence de fuites de mémoire).
6. Rajouter à banque un ensemble de comptes bancaires.
7. Munir la classe banque d'une méthode `appliquerinterets` qui se charge de calculer les intérêts des comptes rémunérés.
8. Écrire une méthode `chercheproprietaire` de recherche d'un propriétaire par son nom ou une partie de son nom (on ne retournera que le premier propriétaire trouvé correspondant à la chaîne passée en paramètre).
9. Écrire une méthode `comptes_numero` prenant comme paramètre un propriétaire et retournant l'ensemble des identifiants des comptes de ce propriétaire. **Attention** si le propriétaire est une personne physique on veut que la méthode retourne aussi les comptes qui ont pour propriétaire un couple dans lequel est présent la personne physique, et les sociétés dont le gérant est la personne physique en question.
10. Écrire une méthode `comptes_de` retournant l'ensemble des comptes d'un propriétaire donné (sans la subtilité de la question précédente).
11. Écrire une méthode `comptes_decouvert` retournant tous les comptes à découvert (dont le montant est inférieur à 0).
12. Écrire une méthode `sommetotale` retournant la somme détenue par un propriétaire donné.
13. *(question optionnelle)* Les clients de la banque peuvent mettre en place des virements automatiques appliqués tous les mois. Chaque virement automatique se fait à partir d'un compte, vers un autre compte, pour un montant donné. Rajouter à `banque` la mémorisation d'un ensemble de virements automatiques et écrire des méthodes pour mémoriser un virement automatique et appliquer un tel virement automatique. Écrire une méthode `appliquervirements`, qui sera appelée tous les mois et qui applique tous les virements.
14. *(question optionnelle)* On veut maintenant mémoriser les opérations effectuées sur les comptes. Ces opérations peuvent être : versement d'espèces, retrait d'espèces, virement vers un autre compte, virement depuis un autre compte. Chaque opération concerne un montant, et peut avoir été refusée ou acceptée (les opérations refusées pour cause de dépassement des valeurs autorisées sont mémorisées elles aussi). Pour chaque compte, on mémorisera les 5 dernières opérations. Modifier les classes précédemment définies pour mémoriser les opérations.

1 Le fonctionnement des « vrais » comptes est évidemment un peu plus compliqué que cela. Ici le calcul des intérêts est fait au moment de l'appel de la méthode `appliquerinterets` alors qu'en réalité il est calculé par quinzaine.