Université de Paris-Cité **Langages à objets avancé**
Master 1

Exercice 1 - Horloges

Remarque : une partie de ce travail est similaire à ce que vous avez déjà fait avec Timestamp au TP1, jouez le jeu : ne regardez pas ce que vous aviez écrit, et constatez vous même si la qualité de ce que vous avez assimilé est satisfaisante.

1. Déclarez simplement une classe `Horloge` qui encapsule en privé des valeurs pour des heures, des minutes et des secondes. Précisez que par défaut l'horloge est à zéro.
2. Surchargez l'opérateur d'affichage pour qu'il soit adapté à votre classe `Horloge`. Testez votre opérateur en faisant en sorte que le constructeur témoigne en affichant l'horloge qu'une construction a eu lieu.
3. Ecrivez le destructeur d'horloges, où vous ferez également un affichage qui réutilise l'opérateur `<<`.
4. Ecrivez une méthode `tick()` qui fait passer une seconde. Pensez qu'après 23H 59M 59S, on passe à 0H 0M 0S. Comme on souhaite être alerté d'un changement de jour, faite en sorte que `tick()` retourne une valeur appropriée. D'ailleurs, avez vous bien fait en sorte qu'à la construction les intervalles de définition soient corrects ?
5. Ecrivez un constructeur de copie pour les Horloges, testez le en faisant en sorte qu'il indique par un message les moments où il est invoqué.
6. Testez cette classe en écrivant un programme qui lit les heures, les minutes et les secondes depuis l'entrée standard (utilisez `cin`), construit l'horloge correspondante, l'affiche, l'incrémente et l'affiche à nouveau en précisant si un changement de jour a eu lieu.
7. Finalement, relisez une fois l'ensemble de votre travail en ajoutant le mot clé **const** partout où une méthode laisse soit l'objet acteur invariant soit les arguments passés invariants.

Exercice 2 [Références/pointeurs/valeurs]

1. Ecrivez rapidement les deux fichiers `.hpp` et `.cpp` d'une classe `BoxInt` qui encapsule un entier. Cette classe aura un "getter" `get()` et un "setter" `set(int)`. Surchargez l'opérateur `<<` pour l'affichage. Vous définirez également un constructeur naturel ainsi qu'un destructeur trivial qui tous deux afficheront un message. (Dans la suite vous pourrez les mettre en commentaires selon le niveau de détail que vous jugerez utile pour observer la trace de l'exécution)
2. Créez un fichier de test qui définit ces trois fonctions

```
void fonction1(BoxInt t) { t.set(345); }

void fonction2(BoxInt *t) { t->set(678); }

void fonction3(BoxInt &t) { t.set(1); }
```

3. Essayez d'anticiper (de tête) le comportement de la séquence suivante, en vous assurant de comprendre les symboles utilisés. Distinguez en particulier les usages de `&`.

```
BoxInt monTest{42};
cout << monTest;

monTest.set(0);
cout << monTest;

fonction1(monTest);
cout << monTest;

fonction2(&monTest);
cout << monTest;

fonction3(monTest);
cout << monTest;
```

   Vérifiez ensuite si vous aviez raison, en exécutant ces instructions dans le `main()` de votre fichier test.

4. Avec votre définition de `BoxInt`, est-il possible de définir la fonction ci-dessous ?

```
void fonction4(const BoxInt &t) { t.set(13); }
```

   Vérifiez votre réponse en compilant/exécutant.

5. même question avec

```
void fonction5(const BoxInt *t) { t -> set(13); }
```

   Vérifiez votre réponse en compilant/exécutant.

6. et avec :

```
void fonction6(BoxInt * const t) { t -> set(13); }
```

   Vérifiez votre réponse en compilant/exécutant.

7. encore une fois, mais avec :

```
void fonction7(BoxInt const * const t) { t -> set(13); }
```

   Vérifiez votre réponse en compilant/exécutant.

8. juste pour être vraiment sûr que vous avez bien écrit les choses, vérifiez que la fonction suivante est acceptée à la compilation :

```
void fonction8(const BoxInt &t) { cout << t.get() << endl; }
```

   Corrigez votre classe si ce n'est pas le cas.

9. On souhaite que chaque `BoxInt` se rappelle du dernier moment où elle a été consultée par `get`. Implémentez cette fonctionnalité en ajoutant un attribut, mais sans changer la signature des méthodes.
   Voir : `https://en.cppreference.com/w/cpp/chrono/c/time`

10. On veut pouvoir connaître le nombre d'instances existantes de `BoxInt` à tout moment. Ajoutez un attribut statique `int` à votre classe et une méthode statique `alive_count()` qui renvoie la valeur de cet entier. Adaptez naturellement le code des constructeurs et du destructeur. Testez votre comptage en créant et supprimant des objets de la classe `BoxInt` avec `new` et `delete` et en affichant ce que renvoie `alive_count()` entre ces opérations dans le fichier test.

11. Imaginez, en particulier la fonction :

```
void un_test(){
        BoxInt un_int{42};
        BoxInt un_autre_int {un_int};
        BoxInt *n = new BoxInt{54};
}
```

et un `main` réduit à

```
int main() {
        un_test();
        // ici combien y a t'il d'instances de BoxInt vivantes ?
            accessibles ? perdues ?
        return EXIT_SUCCESS
}
```

Vous devriez comprendre que vous ne quittez pas le programme en laissant la mémoire propre. Qu'est-il nécessaire de faire ? (et pensez y en règle générale !)

Exercice 3 Cet exercice viens compléter l'exercice 1 qui portait sur les horloges.

— **Dates**
De la même façon, écrivez une classe `Date` (on pourra supposer que les mois ont tous 30 jours), son constructeur, une méthode d'affichage (avec <<) qui donne un résultat sous la forme "10 août 1539" et une méthode pour passer au jour suivant.
Testez cette classe de la même façon que la précédente.

— **RendezVous** En utilisant les deux premières classes :

1. Déclarez une classe `RendezVous` à une certaine date et à une heure.
2. Ecrivez des accesseurs
3. Redéfinissez l'opérateur d'affichage

— **Pointeurs, références, tableaux**

1. Ecrivez une méthode d'horloge qui teste si l'horloge courante est plus petite strictement que l'horloge qui lui est donnée en argument. Soignez la signature de cette méthode pour éviter les copies, et déclarez correctement ce qui est invariant.
2. Dans le fichier qui contient `main`, écrivez une fonction qui prend en argument un tableau d'horloge et sa taille , et qui retourne la référence vers la plus petite horloge de ce tableau.
3. Sur un exemple de 3 valeurs d'horloges de votre choix, récupérez la plus petite avec la fonction précédente, stockez là dans une variable adéquate, et faite la avancer. Vérifiez qu'elle a bien avancé à la fois dans cette variable, et dans le tableau.

Exercice 4 [vector]

Vous n'aurez probablement pas le temps de faire cet exercice pendant la séance. Il n'introduit pas de nouvelle difficulté. Utilisez le pour vous entraîner.

Nous allons programmer ici une implémentation alternative de la classe `vector` de la STL qui représente des tableaux "sans leurs défauts". Puisque les templates n'ont pas encore été vus en cours, nous allons nous focaliser sur des vecteurs d'un seul type : des entiers.

1. Créez les deux fichiers `.hpp` et `.cpp` associés à une classe `Vector` (avec une majuscule pour faire la différence). Cette classe contiendra un `int` qui représentera la taille courante du tableau et un pointeur `int*` vers un tableau d'entiers. Faites en sorte qu'un utilisateur de `Vector` ne puisse pas changer ces attributs directement. Ecrivez un constructeur, un destructeur et et redéfinissez l'opérateur `<<` qui affiche en premier la taille, puis les entiers contenu par votre `Vector` en les séparant par des virgules. On rappelle que l'on crée et supprime des tableaux d'entiers avec les opérations `pointeur = new int[taille]` et `delete[] pointeur`.
2. Écrivez des méthodes `get_at(int)` et `set_at(int,int)` qui respectivement lisent et écrivent dans une case d'un `Vector`. (Nous n'avons pas encore vu les exceptions : retournez simplement 0 après avoir signalé une anomalie éventuelle)
3. Écrivez une méthode `push_back(int)` à votre classe, qui ajoute un entier à la fin du tableau de `Vector`. Votre méthode devra créer un nouveau tableau `int*`, recopier l'ancien dans le nouveau, et supprimer l'ancien. Similairement, écrivez une méthode `push_front(int)` qui ajoute un entier au début du tableau.
4. Écrivez des méthodes `pop_back()` et `pop_front()` qui suppriment et renvoient respectivement le dernier et le premier élément du tableau. (A nouveau esquivez les problèmes de débordements en retournant 0 et en affichant un message)
5. Concevez une procédure qui permet de tester toutes les méthodes de `Vector` définies jusqu'à présent, et écrivez-la dans un fichier `.cpp` de test. On pourra comparez le comportement de `Vector` avec celui de `vector<int>` en répliquant les opérations de la procédure de test sur une instance de cette dernière classe et en comparant les tableaux obtenus. On pourra utiliser `srand()` et `rand()` pour générer des tableaux aléatoires.
6. Copier des `Vector` ou leur contenu, peut être coûteux ou maladroit. Comment s'assurer simplement en C++ qu'aucune de ces opération n'a lieu lors de l'exécution faite par un utilisateur de la classe ? (Il devrait y avoir 2 arguments dans votre réponse)
7. On souhaite pouvoir connaître la mémoire occupée par l'ensemble des objets `Vector` à tout moment. Pour cela, ajoutez à la classe une variable statique représentant l'espace occupé et adaptez les méthodes que vous avez déjà écrites.