TP4: Surcharge d'opérateurs

Introduction On souhaite ici définir une classe permettant de gérer des durées et disposant de tous les opérateurs usuels. Cette durée s'exprimera en heures, minutes, secondes (valeurs entières), et sera mémorisée par ces trois valeurs.

Travail à effectuer

1. Déclarer la classe et lui ajouter les attributs **heures**, **minutes**, **secondes**. Lui donner **un** constructeur ayant des arguments par défaut et permettant la syntaxe suivante :

```
int main()
{
     Duree d1( 3, 4, 5 ); // 3 h 4 mn 5 s
     Duree d2( 3, 4 );    // 3 h 4 mn 0 s
     Duree d3( 3 );       // 3 h 0 mn 0 s
     Duree d4;            // 0 h 0 mn 0 s
}
```

   (Pour l'instant, on fera l'hypothèse que l'utilisateur de la classe ne donnera que des minutes et secondes inférieures à 60.)

2. Ajouter une fonction **amie** définissant l'opérateur **<<** de façon à pouvoir écrire :

```
int main()
{
     Duree d1( 3, 4, 5 ); // 3 h 4 mn 5 s
     cout << "durée=" << d1 << endl;
}
```

   Les lignes ci-dessus devront produire l'affichage :

```
durée = 3h 4mn 5s
```

   Cette fonction aura la signature suivante :

```
std::ostream& operator << ( std::ostream& s, const
   Duree& d );
```

   Ce qui signifie que cette fonction :

   — a deux arguments, le premier étant le flot dans lequel on injectera les valeurs à afficher, le deuxième étant l'objet à injecter.

   — renvoie une référence sur un "flot" **std::ostream** (le type de **cout**). Il faudra donc **renvoyer** le premier argument en fin de fonction.

   Dans le corps de la fonction, on pourra alors injecter dans le flot **s** tout ce qu'on voudra (et notamment les valeurs des attributs de **d**).

```
ostream& operator << ( ostream& s, const Duree& d )
{
     s << ... // compléter
     return s;
}
```

3. Surdéfinition des opérateurs de comparaison

   On souhaite pouvoir comparer des durées, de façon à pouvoir écrire :

```
int main()
{
     Duree d1( 3 );
     Duree d2( 4 );
     if( d1 == d2 )
          cout << "d1 egal d2\n";
     if( d1 < d2 )
          cout << "d1 inférieur à d2\n";
}
```

   Le compilateur ne fournit pas d'implémentation par défaut pour les opérateur de comparaison (**==** et **<**). Il faut donc que vous donniez une définition, via une méthode de classe (un seul argument).

   Tester ensuite avec différentes valeurs.

4. Surdéfinition des opérateurs arithmétiques

   On veut pouvoir manipuler nos durées de la façon la plus intuitive possible, par exemple on pourrait écrire :

```
int main()
{
     Duree d1( 4, 4, 5 );
     Duree d2( 1, 5, 6 );
     cout << "somme=" << d1+d2 << endl;
```

```
cout << "diff=" << d1-d2 << endl;
```

Définir ces deux opérateurs via des méthodes de classe. Ces opérateurs vont renvoyer un nouvel objet **Duree**, déclaré comme variable locale dans la méthode. L'argument (unique) sera transmis sous forme de *référence constante* (comme avec l'opérateur <<).

Factorisation Pour ces deux opérateurs, il sera plus aisé de convertir les deux durées en secondes avant de faire l'addition ou la soustraction puis transformer le résultat au format h./mn./s.

On remarque que ces deux opérations nécessitent cette conversion en h/mn/s après l'opération. Factoriser cette conversion en une méthode privée **void Normalisation(int nbsec)**, qui sera appellée dès que nécessaire (y compris dans le constructeur).

5. Gestion d'erreurs

   L'opérateur de soustraction présente une particularité : suivant les valeurs, il sera possible d'avoir une durée négative, ce qui est proscrit. Il faut donc ajouter un **contrôle** du résultat de l'opération et générer une exception, de façon à pouvoir **intercepter** cette erreur dans le programme :

```
int main()
{
    Duree d1( 1 ), d2( 2 ), d3;
    try
    {
        d3 = d1-d2;
    }
    catch( const string& msg ) // interception des
    {                          // exceptions de type "string"
        cout << "erreur: " << msg << endl;
        throw msg; // on relance l'exception
    }
    catch(...)   // interception de toutes les
    {            // autres exceptions
        cout << "erreur inconnue\n";
        throw; // on relance l'exception
    }
```

   Le lancement d'une exception se fait avec le mot-clé **throw** et en spécifiant l'objet à lancer. L'exécution de la fonction s'interrompt et le contrôle est rendu au niveau supérieur, qui peut alors intercepter cette exception via un bloc **try... catch**. Si le bloc supérieur ne réalise pas cette interception, alors on remonte au niveau supérieur et ainsi de suite. Au final, si aucune interception n'est faite, alors le programme s'interrompt.

   On peut "lancer" tout type d'objet, par exemple :

```
if( a == b ) // lancement d'un objet de type string
  throw std::string("ca ne va pas du tout");
```

   ou bien

```
if( a == b )
{
   BANANE b;
   throw b; // lancement d'un objet de type BANANE
}
```

   Implémenter cette gestion d'erreur et tester, en vérifiant qu'une durée négative provoque bien le lancement d'une exception.

6. Ajout de méthodes

   Ajouter la méthode `int GetNbDays()` qui permettra d'obtenir le nombre de jours correspondant à la durée, en considérant qu'une durée comprise entre 0 et 24h sera de 1 jour.

7. Editer le constructeur de façon à ce que la déclaration suivante :
   `Duree d1( 1, 61, 123 );`
   soit bien convertie en une durée de 2h 3mn 3 s. Vous utiliserez la fonction de normalisation vue précédemment.

8. Amélioration des programmes précédents

   Recharger dans l'IDE le TP1 et ajouter à la classe **Equation** un opérateur d'injection de façon à pouvoir écrire dans un programme :

```
Equation e( 4, 5, 6 );
cout << "equation: " << e << endl;
```

   Ceci devra afficher : `equation: 4 x^2 + 5 x + 6 = 0`