Exercice 28 - Les évènements - Héritage

Afin de pouvoir élaborer un agenda, on désire implémenter un ensemble de classes permettant de gérer des évènements de différents types (anniversaires, rendez-vous, fêtes, jours fériés, etc). Un évènement se passe à une date précise. On identifie un évènement avec un sujet (une description). Certains évènements sont aussi caractérisés par un horaire et une durée. Parmi ces évènements, on distingue les rendez vous avec une ou plusieurs personnes qui ont lieu à un endroit déterminé.
Toutes les classes suivantes seront définies dans l'espace de nom `TIME`. On dispose aussi de classes simples : `Date`, `Duree`, `Horaire` fournies avec le sujet dans les fichiers `timing.h` et `timing.cpp`.
Le polymorphisme étant mis en œuvre, on utilisera la dérivation publique. On définira les constructeurs, destructeurs et accesseurs dans toutes les classes implémentées dans la suite. On fera attention à la gestion des espaces **`private`**, **`protected`** et **`public`** des classes.

Préparation : Créer un projet vide et ajouter trois fichiers `evenement.h`, `evenement.cpp` et `main.cpp`. Définir la fonction principale `main` dans le fichier `main.cpp`.
On suppose qu'un évènement simple qui a lieu un jour est décrit par une date et un sujet. On a donc définit la classe `Evt1j` suivante (à ajouter dans le fichier `evenement.h`):

```
#if !defined(_EVENEMENT_H)
#define _EVENEMENT_H
#include <iostream>
#include <string>
#include "timing.h"

namespace TIME{
 class Evt1j {
 private:
  Date date;
  std::string sujet;
 public:
  Evt1j(const Date& d, const std::string& s):date(d),sujet(s){}
  const std::string& getDescription() const { return sujet; }
  const Date& getDate() const { return date; }
  void afficher(std::ostream& f= std::cout) const {
   f<<"***** Evt ********"<<"\n"<<"Date="<<date<<" sujet="<<sujet<<"\n";
  }
 };
}
#endif
```

evenement.h

Les instructions suivantes (à mettre dans le fichier `main.cpp`) permettent de construire des objets `Evt1j` :

```
#include <iostream>
#include "evenement.h"
int main(){
 using namespace std;
 using namespace TIME;
 Evt1j e1(Date(4,10,1957),"Spoutnik");
 Evt1j e2(Date(11,6,2013),"Shenzhou");
 e1.afficher();
 e2.afficher();
 system("pause");
 return 0;
}
```

main.cpp

S'assurer que le projet compile correctement. Dans cet exercice, on tâchera de mener une approche "compilation séparée". Au fur et à mesure de l'exercice, on pourra compléter la fonction principale en utilisant les éléments créés.

Question 1 - Hiérarchie de classes

Après avoir lu les questions 2 et 3, dessiner un modèle UML représentant la hiérarchie des classes mises en œuvre.

Question 2 - Héritage - Spécialisation

On désire aussi gérer des évènements liés à un jour mais qui comporte aussi un horaire de début et une durée. Un objet de la classe `Evt1jDur` doit permettre de réprésenter de tels évènements.

1. Implémenter la classe `Evt1jDur` qui hérite de la classe `Evt1j`.
2. Ajouter les accesseurs manquants et redéfinir la méthode `afficher`. **Exemple:**

```
#include <iostream>
#include "evenement.h"
int main(){
 using namespace std;
 using namespace TIME;
 Evt1j e1(Date(4,10,1957),"Spoutnik");
 Evt1j e2(Date(11,6,2013),"Shenzhou");
 Evt1jDur e3(Date(11,6,2013),"Lancement de Longue Marche",Horaire(17,38),Duree
    (0,10));
 e1.afficher();
 e2.afficher();
 e3.afficher();
 system("pause");
 return 0;
}
```

main.cpp

Question 3 - Héritage - Spécialisation

On désire aussi gérer des évènements représentant des rendez-vous. Un objet de la classe `Rdv` est un objet `Evt1jDur` avec un lieu et une ou plusieurs personnes.

1. Implémenter la classe `Rdv` (rendez-vous). On utilisera la classe `string` pour ces deux attributs (un seul objet `string` pour toutes les personnes).
2. Ajouter les accesseurs manquants et redéfinir la méthode `afficher`.

Question 4 - Héritage - Construction et destruction

Ajouter un affichage sur le flux `cout` dans les constructeurs des classes `Evt1j`, `Evt1jDur` et `Rdv` en écrivant un message du type "construction d'un objet de la classe X". Définir un destructeur dans chacune de ces classes en y ajoutant un affichage sur le flux `cout` en écrivant un message du type "destruction d'un objet de la classe X". Enfin, exécuter les instructions suivantes :

```
{ // début de bloc
  Rdv e(Date(11,11,2013),"reunion UV",Horaire(17,30),Duree(60),"Intervenants UV","
     bureau");
  std::cout<<"RDV:";
  e.afficher();
} // fin de bloc
```

En déduire la façon dont les différentes parties d'un objet sont construites et détruites.

Exercice 29 - Redéfinition de la duplication par défaut *-Exercice d'approfondissement-*

Redéfinir le constructeur de recopie et l'opérateur d'affectation de la classe `Rdv` (Voir Exercice 28).

Exercice 30 - Les évènements - Polymorphisme

Question 1 - Polymorphisme

Exécuter les instructions suivantes :

```
Evt1j e1(Date(4,10,1957),"Spoutnik");
Evt1j e2(Date(11,6,2013),"Shenzhou");
Evt1jDur e3(Date(11,6,2013),"Lancement de Longue Marche",Horaire(17,38),Duree
   (0,10));
Rdv e4(Date(11,4,2013),"reunion UV",Horaire(17,30),Duree(60),"Intervenants UV","
   bureau");
e1.afficher(); e2.afficher(); e3.afficher(); e4.afficher();
Evt1j* pt1= &e1; Evt1j* pt2=&e2; Evt1j* pt3=&e3; Evt1j* pt4=&e4;
pt1->afficher(); pt2->afficher(); pt3->afficher(); pt4->afficher();
```

1. Qu'observez vous ? Assurez-vous que le polymorphisme est bien mis en œuvre ou faire en sorte qu'il le soit...
2. Surcharger (une ou plusieurs fois) l'opérateur **operator**<< afin qu'il puisse être utilisé avec un objet std ::ostream et n'importe quel évènement.

Question 2 - Polymorphisme et destruction

Ajouter un affichage sur le flux cout dans les constructeurs des classes Evt1j, Evt1jDur et Rdv en écrivant un message du type "construction d'un objet de la classe X". Définir un destructeur dans chacune de ces classes en y ajoutant un affichage sur le flux cout en écrivant un message du type "destruction d'un objet de la classe X". Enfin, exécuter les instructions suivantes :

```
Rdv* pt5= new Rdv(Date(12,11,2013),"reunion UV",Horaire(17,30),Duree(60),"
   Intervenants UV","bureau");
pt5->afficher();
delete pt5;

Evt1j* pt6= new Rdv(Date(12,11,2013),"reunion UV",Horaire(17,30),Duree(60),"
   Intervenants UV","bureau");
pt6->afficher();
delete pt6;
```

Qu'observez vous ? Corriger les problèmes si nécessaire.

Question 3 - Polymorphisme et stockage hétérogène

On veut maintenant disposer d'une classe Agenda qui permet de stocker des évènements.

1. Implémenter une classe Agenda qui pourra permettre de gérer des évènements de tout type (Evt1j, Evt1jDur, Rdv). Pour cela, on utilisera un tableau alloué dynamiquement de pointeurs sur Evt1j. Choisir une certaine taille pour ce tableau, sachant qu'il pourrait être agrandi par la suite si nécessaire. Interdire la duplication (par affectation ou par recopie) d'un objet Agenda.
2. Définir un opérateur Agenda& Agenda::**operator**<<(Evt1j& e) qui permet d'ajouter un évènement dans un objet agenda. Prendre simplement l'adresse de l'évènement passé en argument sans dupliquer l'objet. Quelle type d'association y a t-il entre la classe Agenda et les classes d'évènements ? Compléter le diagramme de classe de la question 1 avec la classe Agenda en conséquence.
3. Définir la fonction **void** afficher(std::ostream& f=std ::cout)**const** qui permet d'afficher tous les évènements d'un objet Agenda.

Exercice 31 - Les évènements - Classes abstraites, Généralisation

On suppose maintenant que certains évènements durent plusieurs jours (conférences, festival, fête). On souhaite alors définir une classe `EvtPj` (évènement de plusieurs jours).
Auparavant, on a donc besoin de généraliser le concept lié à la classe `Evt1j` en introduisant une classe `Evt` qui n'est pas contraint par le nombre de jours. Un objet `Evt1j` est alors un objet `Evt` avec une date et un objet `EvtPj` est un objet `Evt` avec une date de début et une date de fin.

1. Implémenter une classe abstraite `Evt` qui comportera la fonction virtuelle pure `afficher`.
2. Vérifier que la classe `Evt` n'est pas instanciable.
3. Modifier les schémas de dérivation des classes précédentes pour prendre en compte cette nouvelle classe. Remonter l'attribut `sujet` dans la classe `Evt`.
4. Modifier les classes `Evt1j` et `Agenda` et la fonction **`operator<<()`** afin de tenir compte de ces changements (un objet `Agenda` doit maintenant contenir des objet `Evt`).
5. Définir la classe `EvtPj` (évènement de plusieurs jours).
6. Modifier le diagramme de classe en prenant en compte toutes ces modifications.

Exercice 32 - Les évènements - Design Patterns

Question 1 - Design pattern Iterator

Implémenter le design pattern Iterator pour la classe `Agenda` afin de pouvoir parcourir séquentiellement les évènements d'un objet `agenda`. L'itérateur implémenté devra être bidirectionnel (il devra être possible de revenir en arrière dans la séquence).

Question 2 - Design patter Factory Method

Faire en sorte maintenant qu'un objet `Agenda` ait la responsabilité de ses évènements en obtenant une duplication dynamique de l'objet passé en argument. Quelle type d'association y a t-il maintenant entre la classe `agenda` et les classes d'évènements ? Compléter et modifier le diagramme de classe en conséquence.

Question 3 - Design pattern Template Method

Appliquer le design pattern template method en procédant de la manière suivante :

- Déclarer dans la classe `Evt`, la méthode virtuelle pure `string Evt::toString()const` qui renvoie une chaine de caractères décrivant un objet évènement.
- Implémenter cette méthode pour chacune des classes concrètes de la hiérarchie de classe en utilisant la classe standard `stringstream` (voir l'exemple de l'exercice corrigé 24).
- Rendre la méthode `afficher` concrète dans la classe `Evt` et éliminer les anciennes implémentations de cette méthode dans les classes concrètes.

Question 4 - Design pattern Adapter

Lors du développement d'un nouveau système, nous avons besoin d'un objet qui puisse faire l'historique des différents évènements importants qui peuvent survenir (erreurs, authentifications, écritures/lectures dans un fichier). Soit l'interface suivante (qui sera placé dans le fichier `log.h`):

```
#if !defined(LOG_H)
#define LOG_H
#include "timing.h"
#include<iostream>
class Log {
public:
 virtual void addEvt(const TIME::Date& d, const TIME::Horaire& h, const std::string
    & s)=0;
 virtual void displayLog(std::ostream& f) const=0;
};
#endif
```

log.h

La méthode `addEvt` doit permettre d'ajouter un nouvel évènement système caractérisé par une date, un horaire et une description. La méthode `displayLog` doit afficher tous les évènement d'un historique sur un flux `ostream` avec un évènement par ligne sous le format : date - horaire : description.

- Développer une classe concrète `MyLog` qui implémente cette interface en réutilisant au mieux les classes développées précédemment. Pour cela, on appliquera le design pattern Adapter. On fera la question une fois en utilisant un **adaptateur de classe** et une fois en utilisant un **adaptateur d'objet**.
- Compléter le fichier `log.h` en ajoutant une classe d'exception `LogError` qui hérite de la classe d'exception `std::exception`. Dans la méthode `MyLog::addEvt`, déclencher une exception de type `LogError` si l'évènement ajouté est antérieur (date/horaire) au dernier évènement de l'historique (indiquant une probable corruption du système).
- Dans le fichier `main.cpp`, ajouter un bloc **`try-catch`** qui englobe des instructions suceptibles de déclencher des exceptions de type `LogError`. Utiliser un gestionnaire de type `std::exception` pour traiter l'exception.

Exercice 33 - Les évènements - Transtypage et reconnaissance de type à l'exécution

Question 1 - Transtypage dynamique

Corriger le code suivant de façon à ce qu'il compile et qu'il s'exécute sans erreur :

```
Evt1j e1(Date(4,10,1957),"Spoutnik");
Evt1j e2(Date(11,6,2013),"Shenzhou");
Evt1jDur e3(Date(11,6,2013),"Lancement de Longue Marche",Horaire(17,38),Duree
   (0,10));
Rdv e4(Date(11,4,2013),"reunion UV",Horaire(17,30),Duree(60),"Intervenants UV","
   bureau");
Evt1j* pt1= &e1; Evt1j* pt2=&e2; Evt1j* pt3=&e3; Evt1j* pt4=&e4;
Evt1j& ref1=e1; Evt1j& ref2=e2; Evt1j& ref3=e3; Evt1j& ref4=e4;

Rdv* pt=pt1; pt->afficher();
pt=pt2; pt->afficher();
pt=pt3; pt->afficher();
pt=pt4; pt->afficher();

Rdv& r1=ref1; r1.afficher();
Rdv& r2=ref2; r2.afficher();
Rdv& r3=ref3; r3.afficher();
Rdv& r4=ref4; r4.afficher();
```

Question 2 - Transtypage dynamique

Définir un opérateur **`operator<()`** afin de comparer deux évènements dans le temps. Etudier les conversions (up-casting et down-casting) entre objets qui ont un lien de parenté.

Question 3 - Reconnaissance de type à l'exécution **-Exercice d'approfondissement-**

Définir une méthode `agenda::statistiques` qui permet de connaître le nom des différents types d'évènement présents dans un agenda ainsi que le nombre d'occurrence d'évènements pour chaque type d'évènement. On supposera que ces types ne sont pas connus à l'avance (mais on pourra supposer qu'il n'existe pas plus de 10 types différents).