TD4 : 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, fête, jours fériés, etc). Un évènement se passe à une date précise. On identifie un évènement avec un sujet (une description). Certains évènements ont aussi un horaire et une durée. Parmi ces évènements, on distingue les rendez vous avec une ou plusieurs personnes à un lieu déterminé.

Toutes les classes suivantes seront définies dans un espace de nom commun que l'on pourra nommer « `TIME` ». On dispose de classes simples : « `Date` », « `Duree` », « `Horaire` » fournies avec le sujet dans les fichiers `time.h` et `time.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.

On suppose qu'un évènement est lié à un jour et est donc décrit par une date et un sujet. On a donc définit la classe `ev_1j` suivante :

```
namespace TIME{
	class ev_1j {
	protected:
		string _objet;
		Date _date;
	public:
		ev_1j(int j, int m, int a, const string& obj):
							_date(j,m,a),_objet(obj){}
		const string& get_description() const { return _objet; }
		const Date& get_date() const { return _date; }
		virtual void afficher(ostream& f= cout) const {
			f<<"***** evt ********"<<"\n";
			f<<"Date="<<_date<<" objet="<<_objet<<"\n";
		}
	};
}
```

Hiérarchie de classes

Question 1

Lire le sujet en entier et dessiner un modèle UML représentant les classes mises en œuvre et leurs relations.

Héritage

Question 2

On désire aussi gérer des évènements liés à un jour mais qui comporte aussi un horaire de début et une durée.

a) Implémenter la classe « `Ev_1j_dur` » qui hérite
b) Ajouter les accesseurs manquants et redéfinir la fonction membre « `Afficher` ».

Question 3

a) Implémenter la classe « `Rdv` » (rendez-vous…) qui est un « `Ev_1j_dur` » avec un lieu et une ou plusieurs personnes. On utilisera la classe `string` pour ces deux attributs (un seul objet `string` pour toutes les personnes).
b) Ajouter les accesseurs manquants et redéfinir la fonction membre « `Afficher` ».

Polymorphisme

Question 4

Surcharger (une ou plusieurs fois) l'opérateur d'insertion « `operator<<` » afin qu'il puisse être utilisé avec un objet « `std::ostream` » et n'importe quel évènement. S'assurer que le polymorphisme est bien mis en œuvre ou faire en sorte qu'il le soit...

Question 5

a) Implémenter une classe `Agenda` qui pourra permettre de gérer des évènements de tout type (`Ev_1j`, `Ev_1j_dur`, `Rdv`). Pour cela, on utilisera un tableau de pointeurs sur « `Ev_1j_dur` ».
b) Définir un opérateur « `void operator<<(Ev_1j& e)` » qui permet d'ajouter un évènement dans l'Agenda. Prendre simplement l'adresse de l'évènement passé en argument sans dupliquer l'objet. Quelle type d'association y a t-il entre un agenda et des évènements ?
c) Définir la fonction « `void Afficher(std::ostream& f=std ::cout) const` » qui permet d'afficher tous les évènements de l'Agenda.

(S'il reste du temps)

Question 6

Faire en sorte maintenant que l'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 un agenda et des évènements ?

Question 7

Implémenter le design pattern Iterator pour la classe Agenda afin de pouvoir parcourir séquentiellement ses évènements.