Exercice 26 - 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 27 - Redéfinition de la duplication par défaut *-Exercice optionnel d'approfondissement-*

Redéfinir le constructeur de recopie et l'opérateur d'affectation de la classe `Rdv` (Voir Exercice 26).

Exercice 28 - 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.

Exercice 29 - 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 conteneur standard dans lequel son stockera des pointeurs sur `Evt1j`. 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.
3. Quel 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.
4. Définir la fonction `void afficher(std::ostream& f=std ::cout) const` qui permet d'afficher tous les évènements d'un objet `Agenda`.

Exercice 30 - 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 31 - 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.

Exercice 32 - Les évènements - Transtypage dynamique

Question 1 - Transtypage - mécanisme

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 - application

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.

Exercice 33 - Reconnaissance de type à l'exécution *-Exercice d'approfondissement optionnel-*

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).

Exercice 34 - Les évènements - Design Patterns - II

Question 1 - 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`.
- Rendre la méthode `afficher` concrète dans la classe `Evt` et éliminer les anciennes implémentations de cette méthode dans les classes concrètes.

Question 2 - 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 35 - Les conteneurs

Dans cet exercice, il s'agit de développer un ensemble de classes qui permettent de stocker des objets de n'importe quel type (tableaux d'objets, liste chaînée d'objets, pile d'objets, etc). Les objets peuvent par exemple être des entiers, des réels, des fractions, des évènements, etc. Le terme générique *conteneur* est utilisé pour désigner les classes qui permettent de contenir d'autres objets. On souhaite que chaque conteneur implémenté ait un mode d'utilisation commun et donc une interface commune obligatoire à tous les conteneurs.
On appelle *taille d'un conteneur* le nombre d'objets qu'il contient. Un conteneur est vide lorsqu'il ne contient pas d'objet. On considère que les objets sont indicés à partir de 0. Le premier objet d'un conteneur sera donc le 0^e objet du conteneur.
Dans la suite, on appelle `T`, le type des objets contenus dans les conteneurs. L'interface commune à chaque conteneur est la suivante :

- **unsigned int** `size()` **const;** qui renvoie la taille du conteneur.
- **bool** `empty()` **const;** qui renvoie vrai si le conteneur est vide et faux sinon.
- `T& element(`**unsigned int** `i);` qui renvoie une référence sur le i^e élément d'un conteneur.
- **const** `T& element(`**unsigned int** `i)` **const;** qui renvoie une référence **const** le i^e élément d'un conteneur.
- `T& front();` qui renvoie une référence **const** sur le premier objet contenu dans le conteneur.
- **const** `T& front()` **const;** qui renvoie une référence **const** sur le premier objet contenu dans le conteneur.
- `T& back();` qui renvoie une référence sur le dernier objet contenu dans le conteneur.
- **const** `T& back()` **const;** qui renvoie une référence **const** sur le dernier objet contenu dans le conteneur.
- **void** `push_back(`**const** `T& x);` qui ajoute un objet `x` au conteneur après le dernier objet.
- **void** `pop_back();` qui retire le dernier objet du conteneur.
- **void** `clear();` qui retire tous les objets du conteneur.

Préparation : Créer un projet vide et ajouter deux fichiers `container.h` et `main.cpp`. Les situations exceptionnelles seront gérées en utilisant la classe d'exception suivante (à recopier dans le fichier `container.h`) :

```
#if !defined(_Container_T_H)
#define _Container_T_H
#include<string>
#include<stdexcept>

namespace TD {
class ContainerException : public std::exception {
protected :
 std::string info;
public:
 ContainerException(const std::string& i="") noexcept :info(i){}
 const char* what() const noexcept { return info.c_str(); }
 ~ContainerException()noexcept{}
};
}
#endif
```

container.h

Définir la fonction principale `main` dans le fichier `main.cpp`. S'assurer que le projet compile correctement.
Dans la suite, vous déclarerez et définirez chaque constructeur, chaque destructeur, chaque attribut et chaque méthode (de l'interface obligatoire) partout où cela est nécessaire. Vous définirez aussi les constructeurs de recopie et les opérateurs d'affectation nécessaires.

Question 1

Analyser et modéliser les deux classes du problème dans un diagramme de classe. Implémenter la classe abstraite `Container` modèle de tous les autres conteneurs en exploitant au mieux le design pattern *"template method"* pour utiliser le moins de méthodes virtuelles pures possible.

Question 2

Implémenter une classe `Vector` qui sera basée sur le modèle `Container`. Cette classe utilisera un attribut de type `T*` qui pointera sur un tableau de `T` alloué dynamiquement pour composer ses éléments. Pour cela, on suppose que le type `T` dispose d'un constructeur sans argument.

Le constructeur "principal" prendra en argument la taille initiale du tableau et la valeur avec laquelle les objets initialement présents dans le tableau doivent être initialisés. On étudiera éventuellement la possibilité d'avoir des arguments par défaut. Surcharger en plus l'opérateur **`operator`**`[]` qui permettra de *modifier* ou de *lire* la valeur d'un élément particulier du tableau.

Exercice 36 - Conteneurs - Design pattern Adaptateur et Stratégie

En utilisant astucieusement le design pattern *"adapter"*, implémenter une classe `Stack` qui ne devra avoir comme seule interface possible que les méthodes suivantes :

- `bool empty()const;`
- `void push(const T& x);` qui empile un objet dans la pile.
- `void pop();` qui dépile le dernier élément empilé de la pile.
- `unsigned int size()const;`
- `T& top();` qui renvoie une référence sur le dernier objet empilé de la pile
- `const T& top()const;`
- `void clear();`

On réfléchira à la possibilité de pouvoir "adapter" n'importe quel conteneur pour implémenter cette classe (design pattern *"stratégie"*). On fera cet exercice deux fois : une fois en utilisant un *adaptateur de classe*, une fois en utilisant un *adaptateur d'objet*. On dessinera les diagrammes de classe correspondants.

Exercice 37 - Conteneurs - Design pattern Iterator et algorithmes

Question 1

Implémenter le design pattern "itérateur" en créant le type `iterator` pour les classes `Vector` et `Stack` :

- Pour accéder à l'élément désigné par un itérateur, on utilisera l'opérateur **`operator`**$\star$.
- Pour qu'un itérateur désigne l'élément suivant, on lui appliquera l'opérateur **`operator`**++.
- Afin de comparer deux itérateurs, on surchargera les opérateurs **`operator`**== et **`operator`**!= : on suppose que deux itérateurs sont égaux s'ils désignent le même élément.
- Pour les classes `Vector` et `Stack`, on implémentera la fonction `begin()` qui renvoie un itérateur désignant le premier élément.
- Pour les classes `Vector` et `Stack`, on implémentera aussi la fonction `end()` qui renvoie un itérateur désignant l'élément (fictif) qui suit le dernier élément, c'est à dire l'itérateur que l'on obtient si on applique l'opérateur ++ sur un itérateur désignant le dernier élément.
- Pour le type `Stack::iterator`, préciser les différentes possibilités d'implémentation.
- Avec un simple copier/coller et quelques modifications, on implémentera aussi un type `const_iterator` ainsi que les méthodes `begin()` et `end()` correspondantes.

Question 2

Implémenter la fonction `minimum_element` qui prend en arguments deux itérateurs `it1` et `it2` de n'importe quel conteneur et qui permet de renvoyer un itérateur désignant l'élément minimum dans le conteneur entre les itérateurs `it1` et `it2` (`it2` non compris), par rapport à l'opérateur **`operator`**<. On supposera pour cela que cet opérateur a été surchargé pour le type d'élément contenu dans le conteneur.

Question 3

Implémenter la fonction `minimum_element` qui prend en arguments deux itérateurs `it1` et `it2` de n'importe quel conteneur, ainsi qu'un prédicat binaire définissant un ordre sur les éléments (design pattern *"Strategy"*). La fonction permet de renvoyer un itérateur désignant l'élément minimum par rapport au prédicat binaire dans le conteneur entre les itérateurs ‘`it1` et `it2` (`it2` non compris). Le prédicat binaire doit renvoyer **`true`** ou **`false`**. Il pourra être soit une fonction prenant en arguments deux objets du type de ceux contenus dans le conteneur, soit un *objet fonction* dont l'opérateur **`operator`**`()` prend en arguments deux objets du type de ceux contenus dans le conteneur, soit une lambda-expression équivalente.

Exercice 38 - Pour aller plus loin avec les conteneurs...

-Exercice d'approfondissement optionnel-

Question 1

Implémenter une classe `List` qui sera une liste doublement chaînée.
Le constructeur "principal" prendra en argument la taille initiale de la liste et la valeur avec laquelle les objets initialement présents dans la liste doivent être initialisés. On étudiera éventuellement la possibilité d'avoir des arguments par défaut.
Définir les fonctions **`void`** `push_front(`**`const`** `T& x)` et **`void`** `pop_front()` qui ajoute ou retire un élément en tête de liste.
Définir la fonction **`bool`** `remove(`**`const`** `T& x)` qui retire le premier élément de la liste qui a la valeur `x`. La fonction renverra **`true`** si l'opération réussit et **`false`** sinon (si `x` n'existe pas dans la liste).

Question 2

On suppose maintenant que le type `T` ne dispose pas forcément d'un constructeur sans argument. Modifier votre classe `Vector` de manière à prendre cet aspect en compte.
Pour cela, on utilisera la classe standard `allocator` du C++ (voir poly) qui permet de séparer l'allocation et la désallocation d'une zone mémoire de la construction et de la destruction d'objets (ou de tableaux d'objets) dynamique.
Le constructeur "principal" prendra en argument la taille initiale du tableau et la valeur avec laquelle les objets initialement présents dans le tableau doivent être initialisés. On étudiera éventuellement la possibilité d'avoir des arguments par défaut.
On appellera *"capacité d'un tableau"* le nombre maximum d'objets qu'il peut contenir avant de devoir refaire une réallocation. Initialement, cette capacité est égale à la taille du tableau. Cependant, on pourra changer la capacité (sans pour autant changer la taille) grâce à la méthode **`void`** `reserve(`**`unsigned int`** `n)`. Cette méthode n'a une action que si `n` est strictement supérieur à la taille du tableau (dans le cas contraire, la méthode ne fait rien). Le tableau dispose alors d'une réserve supplémentaire qu'il peut utiliser lorsque le nombre d'éléments augmente (par ex avec la méthode `push_back`) sans devoir pour autant faire une réallocation. Implémenter la méthode **`unsigned int`** `capacity()`**`const`** permettant de connaître la capacité du vecteur.
On implémentera aussi la méthode **`void`** `resize(`**`unsigned int`** `t=0,` **`const`** `T& initialize_with=T());` qui permet de changer le nombre d'éléments du tableau. Si la taille diminue, la capacité ne change pas : les objets qui doivent disparaitre du tableau sont "détruits" mais les cellules qui ne sont plus utilisées sont gardées en réserve (elles ne sont pas désallouées). Si la taille augmente, les nouveaux emplacements utilisées sont initialisées avec la valeur `initialize_with`. Si la nouvelle taille est inférieure à la capacité il n'y a pas de réallocation. Si la nouvelle taille est supérieure à la capacité, un nouveau tableau est réalloué et la capacité devient égale à la nouvelle taille.
De même, la méthode `push_back` ne provoque pas de réallocation tant que la capacité le permet. La méthode `clear` ne provoque pas non plus une désallocation mais seulement une destruction des objets du tableau.

Exercice 39 - Graphes et STL et patrons

Question 1

Après l'exercice 25, on veut développer une classe plus flexible qui permet d'utiliser n'importe quel type pour représenter des sommets. Pour cela, on utilise maintenant un attribut de type `map<Vertex,set<Vertex> >` où `Vertex` est un paramètre de type pour représenter la liste d'adjacence. Un sommet est alors la clé qui permet d'accéder à un ensemble de sommets adjacents.
Implémentez la classe paramétrée dont l'interface est la suivante :

```
#include<map>
#include<set>
#include<string>
#include<iostream>
using namespace std;
template<class Vertex>
class GraphG {
 map<Vertex,set<Vertex> > adj;
 string name;
public:
 GraphG(const string& n);
 const string& getName() const;
 unsigned int getNbVertices() const;
 unsigned int getNbEdges() const;
 void addVertex(const Vertex& i);
 void addEdge(const Vertex& i, const Vertex& j);
 void removeEdge(const Vertex& i, const Vertex& j);
 void removeVertex(const Vertex& i);
 void print(ostream& f) const;
};

template<class V> ostream& operator<<(ostream& f, const GraphG<V>& G);
```

Les deux méthodes `addVertex` et `addEdge` permettent d'ajouter des sommets et des arcs librement. L'ajout d'un arc entre un ou des sommets qui n'existent pas encore provoque leur création. La suppression d'un sommet provoque la suppression de tous les arcs liés à ce sommet.

Question 2 - Exercice d'approfondissement optionnel

Créez les types `vertex_iterator` et `successor_iterator` implémentant le design pattern *iterator* :

- Un objet `vertex_iterator` permet de parcourir séquentiellement tous les sommets du graphe.
- Un objet `successor_iterator` permet de parcourir séquentiellement tous les successeurs d'un sommet donné.

S'inspirer de l'exemple suivant pour l'interface de ces types. Pour créer ces types, on utilisera des *adaptateurs de classe* des types `map<Vertex,set<Vertex> >::const_iterator` et `set<Vertex>::const_iterator`.

Question 3 - Exercice d'approfondissement optionnel

Utilisez l'algorithme standard `std::for_each` avec un objet fonction (voir rappels dans le poly) pour implémenter la fonction `operator<<(ostream&, const GraphG<V>&)` (même si c'est inutilement compliqué !).

Exemple :

```
try{
  GraphG<char> G2("G2");
  G2.addVertex('a'); G2.addVertex('b'); G2.addEdge('a','c');
  G2.addEdge('a','d'); G2.addEdge('d','e'); G2.addEdge('e','b');
  cout<<G2;
  cout<<"vertices of G2 are: ";
  for(GraphG<char>::vertex_iterator it=G2.begin_vertex();
      it!=G2.end_vertex(); ++it) cout<<*it<<" ";
  cout<<"\nsuccessors of a: ";
  for(GraphG<char>::successor_iterator it=G2.begin_successor('a');
```

```
      it!=G2.end_successor('a'); ++it){ std::cout<<*it<<" "; }
  GraphG<string> G3("Pref");
  G3.addEdge("LO21","IA01"); G3.addEdge("IA02","IA01"); G3.addEdge("IA01","NF17");
  G3.addEdge("IA02","NF16"); G3.addEdge("NF93","NF16");
  cout<<G3;
}catch(exception e){ std::cout<<e.what()<<"\n"; }
```

Affichage obtenu :

```
graph G2 (5 vertices and 4 edges)
a:c d
b:
c:
d:e
e:b
vertices of G2 are: a b c d e
successors of a: c d
graph Pref (6 vertices and 5 edges)
IA01:NF17
IA02:IA01 NF16
LO21:IA01
NF16:
NF17:
NF93:NF16
```