TP2. Mise en oeuvre de la classe générique Graphe<S,T> (2h)

1. Mise en oeuvre de la classe (non générique) GElement

GElement (pour Graphe-Element) est la classe de base dont héritent les classes *Sommet<T>* et *Arete<S,T>*.
Elle permet de factoriser les notions communes à ces deux classes. Ici, la seule notion commune à Sommet<T> et Arete<S,T> est une clef primaire. *GElement* est donc une classe dont le seul attribut est un entier *clef* servant de clef primaire.
Il faut donc :
Ecrire la classe *GElement* ainsi définie. Pour simplifier, le membre *clef* peut être public.
Munir *GElement* d'un constructeur, d'un opérateur de conversion en *string* (fonction membre dont la signature est : ***operator*** *string () const*) et de l'opérateur << d'écriture sur un flux (cette dernière méthode est une fonction ordinaire, elle n'est ni membre, ni *friend*, elle peut avantageusement se servir de l'opérateur de conversion en *string*).
Il est inutile d'écrire destructeur, constructeur de copie ou getters et setters.
Ecrire (dans un fichier *TestGraphe.cpp*) une fonction *main*() pour tester les fonctionnalités des classes *GElement*, *Sommet<T>*, *Arete<S,T>* et *Graphe<S,T>*. Tester l'unique constructeur de *GElement*, l'opérateur de conversion en *string* et l'opérateur <<.

2. Mise en oeuvre de la classe générique Sommet<T>

La classe générique *Sommet<T>* représente un sommet dans un graphe.
Elle dérive de *GElement* et contient deux attributs :
degre : un entier représentant le degré du sommet
v : instance de la classe générique *T,* représentant l'information associée au sommet.
Ecrire la classe *Sommet<T>* ainsi définie. Pour simplifier, *degre* et *v* peuvent être publics.
Munir *Sommet<T>* d'un constructeur, d'un opérateur de conversion en *string* (fonction membre dont la signature est : ***operator*** *string () const*) et de l'opérateur << d'écriture sur un flux (cette dernière méthode est une fonction ordinaire, elle n'est ni membre, ni *friend*, elle peut avantageusement se servir de l'opérateur de conversion en string).
Par défaut, un sommet est créé isolé, donc de degré nul.
Nous faisons l'hypothèse que la classe *T* est munie d'un opérateur << d'écriture sur un flux.
Il est inutile d'écrire destructeur, constructeur de copie ou getters et setters. Tester constructeur, operateur string et opérateur << pour les classes *Sommet<double>* et *Sommet<string>*.

3. Mise en oeuvre de la classe générique Arete<S,T>

La classe générique *Arete<S,T>* représente une arête dans un graphe.
Elle dérive de *GElement* et contient trois attributs :
debut : de type *Sommet<T>** représentant un pointeur sur l'extrémité initiale de l'arête
fin : de type *Sommet<T>** représentant un pointeur sur 1'extrémité finale de l'arête
v : instance de la classe générique *S,* représentant l'information associée à l'arête.
Ecrire la classe *Arete<S,T>* ainsi définie. Pour simplifier, *debut*, *fin* et *v* peuvent être publics.
Munir *Arete<S,T>* d'un constructeur, d'un opérateur de conversion en *string* et de l'opérateur << d'écriture sur un flux.

Nous faisons l'hypothèse que la classe *S* est munie de l'opérateur << d'écriture sur un flux.
Il est suffisant, pour la conversion en *string* d'une arête, d'indiquer les clefs primaires des sommets *debut* et *fin*.
Il est inutile d'écrire destructeur, constructeur de copie ou getters et setters.
L'unique constructeur de *Arete<S,T>* ne fait pas de copie de ses arguments de type *Sommet<T>**. Tester constructeur, operateur string et opérateur << pour la classe *Arete<string,char>*.

Munir la classe *Arete<S,T>* de la méthode suivante :

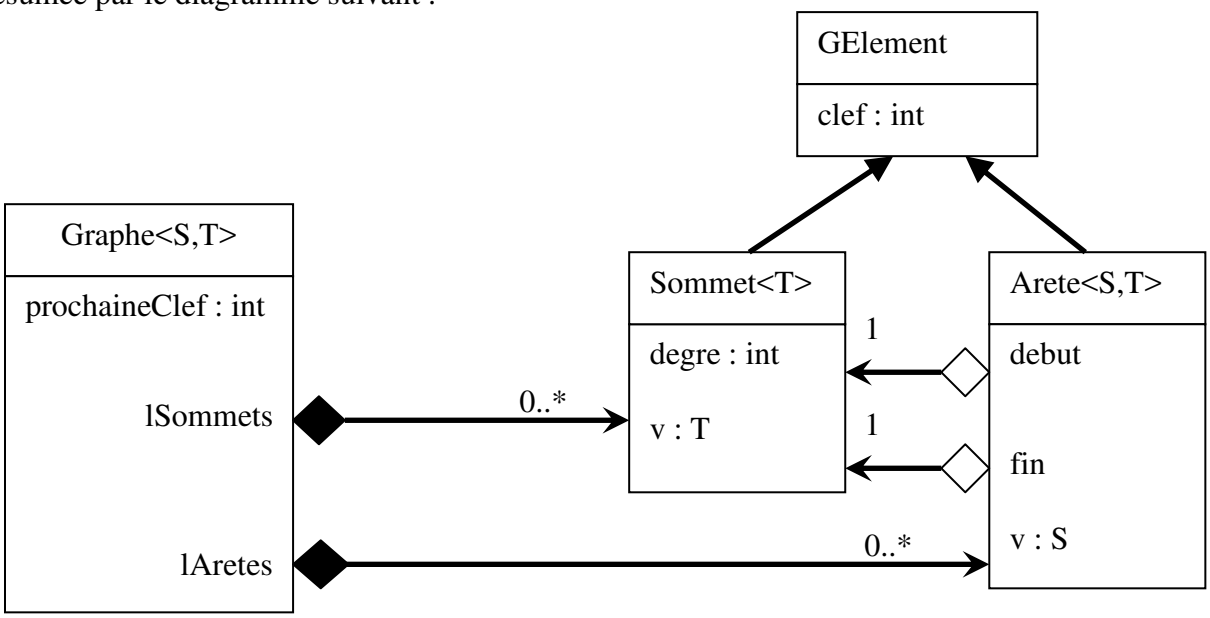
```
bool estEgal( const Sommet<T> * s1, const Sommet<T> * s2) const;
```

qui vérifie si l'arête est égale à l'arête s1 - s2.
Renvoie `true` si *this == s1-s2 ou si *this == s2-s1
Renvoie `false` sinon.
Tester la méthode.

4. Mise en oeuvre de la classe générique Graphe<S,T>

La classe générique *Graphe<S,T>* représente un graphe quelconque. *T* est la nature des informations associées aux sommets et *S* celle des informations associées aux arêtes.
Un graphe est défini par une liste de sommets, par une liste d'arêtes et par un générateur de clefs primaires.
La classe Graphe<S,T> est donc munie de trois attributs :
lSommets : de type *PElement< Sommet<T> > **, qui contient la liste des sommets.
lAretes : de type *PElement<Arete<S,T>> **, qui contient la liste des arêtes.
prochaineClef : de type *int* qui définit la clef primaire du prochain élément (sommet ou arête) qui sera inséré dans le graphe. *prochaineClef* est incrémenté à chaque insertion d'un élément.

L'organisation des classes *GElement*, *Sommet<T>*, *Arete<S,T>* et *Graphe<S,T>* peut être résumée par le diagramme suivant :



4.1 Ecrire la définition de la classe Graphe <S,T>

Ecrire la définition de la classe avec ses attributs.

4.2 Munir la classe Graphe <S,T> du constructeur suivant :

`Graphe();`
qui crée un graphe vide. Initialise de façon cohérente les attributs *prochaineClef*, *lSommets* et *lAretes*.
Note : On peut aussi mettre en oeuvre un constructeur de copie, un destructeur et l'opérateur d'affectation =. Ces 3 méthodes sont nécessaires à une gestion rigoureuse de la mémoire. C'est ici obligatoire car la classe comporte des membres alloués dynamiquement par ses soins. Cependant, par manque de temps, nous ne le ferons pas pendant ce TP.

4.3 Munir la classe Graphe<S,T> de la méthode suivante :

```
Sommet<T> * creeSommet(const T & info);
```
qui crée, dans le graphe, un sommet isolé muni de l'information *info*. Un pointeur sur le sommet créé est retourné.

4.4 Munir la classe Graphe<S,T> de la méthode suivante :

```
Arete<S,T> * creeArete( Sommet<T> * debut, Sommet<T> * fin, const S & info);
```

qui crée, dans le graphe, une arête reliant les sommets *debut* et *fin*. A l'appel, on suppose que *debut* et *fin* sont déjà contenus dans le graphe. L'arête créée contient l'information *info*. La méthode ne fait pas de copie de *debut* ou de *fin*. Les degrés de debut et de fin sont mis à jour par la méthode. Un pointeur sur l'arête créée est retourné.

4.5 Munir la classe Graphe<S,T> de la méthode suivante :

```
int nombreSommets() const;
```

qui renvoie le nombre de sommets contenus dans le graphe.

4.6 Munir la classe Graphe<S,T> de la méthode suivante :

```
int nombreAretes() const;
```

qui renvoie le nombre d'arêtes contenues dans le graphe.

4.7 Munir la classe Graphe<S,T> des opérateurs suivants :

opérateur de conversion en *string* et opérateur << d'écriture sur un flux.

4.8 Munir la classe Graphe<S,T> de la méthode suivante :

```
Arete<S,T> * getAreteParSommets( const Sommet<T> * s1, const Sommet<T> * s2) const;
```
qui retourne l'arête s'appuyant sur *s1* et *s2* (on rappelle qu'une arête n'est pas orientée).
Retourne `NULL` si l'arête *s1* - *s2* n'existe pas dans le graphe.

4.9 Munir la classe Graphe<S,T> de la méthode suivante :

```
PElement< pair< Sommet<T> *, Arete<S,T>* > >  *
adjacences(const Sommet<T> * sommet) const;
```

qui construit, pour le sommet *sommet*, la liste de toutes les associations (voisin de *sommet*, arête incidente en *sommet*).
De cette liste, on peut donc déduire facilement, par exemple, la liste des voisins de *sommet* ou encore la liste des arêtes adjacentes à *sommet*.
Cette liste est essentielle au déroulement de A*.

4.10 Tester les fonctionnalités de Graphe<S,T>

Tester les fonctionnalités de Graphe<S,T> **au fur et à mesure** qu'elles sont écrites.


Créer un Graphe<string,char> à 5 sommets, dont un isolé, puis tester dans l'ordre le constructeur de graphe vide, l'opérateur <<, les méthodes *creeSommet()*, *creeArete()*, *getAreteParSommets* puis enfin *adjacences()*.

Solution du TP2.

Mise en oeuvre de la classe générique Graphe<S,T>

---------- d'abord le fichier GElement.h :

1. Classe GElement

```
/**
Représente la classe de base des éléments d'un graphe qui peuvent
être des sommets ou des arêtes.

A ce niveau, on ne peut que définir que la clef d'un élément.

En effet tout élément d'un graphe est défini par une clef (c-à-d une
valeur non nulle et unique)
*/

class GElement
{
public:
int clef;
GElement(const int clef);
operator string() const;
};

ostream & operator << (ostream & os, const GElement& gElement);
```

---------- puis le fichier GElement.cpp :

```
#include "GElement.h"

GElement::GElement(const int clef):clef(clef){}

GElement::operator string() const
{
ostringstream oss;
oss << "clef = " << clef;
return oss.str();
}

ostream & operator << (ostream & os, const GElement& gElement)
{
return os << (string) gElement;
}
```

2. Classe Sommet<T>

```
/**
Sommet d'un graphe en général
v est l'information associée au sommet : elle dépend de
l'application du graphe
*/
template <class T>
class Sommet : public GElement
{
public:
int degre;
T v;
Sommet(const int clef, const T & v):GElement(clef),degre(0),v(v){}

operator string () const;
};

template <class T>
Sommet<T>::operator string () const
{
ostringstream oss;

oss <<"Sommet("<<endl;
oss << GElement::operator string()<<endl;
oss<<"degré = " << degre << endl;
oss <<"v = " <<  v << endl;
oss<<")";
return oss.str();
}

template <class T>
ostream & operator << (ostream & os, const Sommet<T> & sommet)
{
return os << (string) sommet;
}
```

3. Classe Arete<S,T>

```
/**
représente une arête d'un graphe en général.
Une arête est définie par un sommet-début et par un sommet-fin et
par une information v.

On ne connait pas la nature de v à ce niveau (v pourrait contenir
une couleur, une longueur, un nom, etc.)

T est la nature de l'information portée par un sommet et
S est la nature de l'information portée par une arête
*/

template <class S, class T>
class Arete : public GElement
{
public:
Sommet <T> *debut, *fin;
S v;
Arete(int clef, Sommet<T> * debut, Sommet<T> * fin, const S & v):
GElement(clef),debut(debut),fin(fin),v(v){}

operator string () const;

/**
 * vérifie si *this s'appuie sur s1 et s2
 *
 * DONNEES : *this, s1, s2
 *
 * RESULTATS : true si *this s'appuie sur s1 et s2 c'est-à-dire si
(début == s1 et fin == s2) ou (début == s2 et fin == s1), false
sinon
 *
 * */
bool estEgal( const Sommet<T> * s1, const Sommet<T> * s2) const;

};

template <class S, class T>
Arete<S,T>::operator string () const
{
ostringstream oss;

oss <<"Arete  ("<< endl;
oss << GElement::operator string()<<endl;
oss << "clef debut = " << debut->clef<< endl;
oss << "clef fin = " << fin->clef << endl;
```

```
oss << "v = " << v << endl;
oss << ")";
return oss.str();
}

template <class S, class T>
ostream & operator << (ostream & os, const Arete<S,T> & arete)
{
return os << (string) arete;
}

**
 * vérifie si *this s'appuie sur s1 et s2
 *
 * DONNEES : *this, s1, s2
 *
 * RESULTATS : true si *this s'appuie sur s1 et s2 c'est-à-dire si
(début == s1 et fin == s2) ou (début == s2 et fin == s1), false
sinon
 *
 * */
template <class S, class T>
bool Arete<S,T>::estEgal( const Sommet<T> * s1, const Sommet<T> * s2)
const
{
return (s1 == debut && s2 == fin) || (s1 == fin && s2 == debut);
}
```

4. Classe Graphe<S,T>

4.1 Définition

```
template <class S, class T>
class Graphe
{
protected:
int prochaineClef;
public:

PElement< Sommet<T> > * lSommets; // liste de sommets
PElement< Arete<S,T> > * lAretes; // liste d'arêtes

/**
 * crée un graphe vide
 * */
Graphe();

/**
 * constructeur de copie obligatoire car la classe comporte une
partie dynamique
 * */
Graphe(const Graphe<S,T> & graphe);

/**
 * opérateur = obligatoire car la classe comporte une partie
dynamique
 * */
const Graphe<S,T> & operator = (const Graphe<S,T> & graphe);

/**
 * destructeur obligatoire car la classe comporte une partie
dynamique
 * */
~Graphe();

int nombreSommets() const;
int nombreAretes() const;

/**
 * crée un sommet isolé
 * */
Sommet<T> * creeSommet(const T & info);

/**
 * crée une arête joignant les 2 sommets debut et fin
 *
```

```
 * * met à jour les champs degré de debut et de fin
 * */
Arete<S,T> *
creeArete( Sommet<T> * debut, Sommet<T> * fin, const S & info);

/**
recherche la liste des paires (voisin, arête) adjacentes de sommet
dans le graphe
*/
PElement< pair< Sommet<T> *, Arete<S,T>* > >  *
adjacences(const Sommet<T> * sommet) const;

operator string() const;

/**
 * cherche l'arête s1 - s2 ou l'arête s2 - s1 si elle existe
 *
 * DONNEES : s1 et s2 deux sommets quelconques du graphe
 * RESULTATS : l'arête s'appuyant sur s1 et s2 si elle existe, NULL
sinon
 *
 * */
Arete<S,T> * getAreteParSommets( const Sommet<T> * s1, const
Sommet<T> * s2) const;


//---------------------- Graphe ----------------------------------
};
```

4.2 Constructeurs, destructeurs et opérateur =

```
/**
 * crée un graphe vide
 *
 * */
template <class S, class T>
Graphe<S,T>::Graphe():prochaineClef(0),lAretes(NULL),lSommets(NULL){}

template <class S, class T>
Graphe<S,T>::Graphe(const Graphe<S,T> & graphe)
{
throw Erreur("pas encore écrit : reste à faire");
}

template <class S, class T>
const Graphe<S,T> & Graphe<S,T>::operator = (const Graphe<S,T> &
graphe)
{
```

```
throw Erreur("pas encore écrit : reste à faire");
}

template <class S, class T>
Graphe<S,T>::~Graphe()
{
PElement< Arete<S,T>>::efface2(this->lAretes);
PElement<Sommet<T> >::efface2(this->lSommets);
}
```

4.3 Création d'un sommet isolé

```
/**
 * crée un sommet isolé
 *
 * */
template <class S, class T>
Sommet<T> * Graphe<S,T>::creeSommet( const T & info)
{
Sommet<T> * sommetCree = new Sommet<T>( prochaineClef++,info);
lSommets = new PElement< Sommet<T> >( sommetCree, lSommets);

return sommetCree;
}
```

4.4 Création d'une arête

```
/**
 * crée une arête joignant les 2 sommets debut et fin
 *
 * met à jour les champs degre de debut et de fin
 * */
template <class S, class T>
Arete<S,T> * Graphe<S,T>::creeArete( Sommet<T> * debut, Sommet<T> *
fin, const S & info)
{
Arete<S,T> * nouvelleArete;

// ici tester que les 2 sommets sont bien existants dans le graphe
if (! PElement< Sommet<T> >::appartient(debut,lSommets) ) throw
Erreur("début d'arête non défini");
if (! PElement< Sommet<T> >::appartient(fin,lSommets)) throw
Erreur("fin d'arête non définie");

nouvelleArete = new Arete<S,T>( prochaineClef++, debut, fin,  info);

lAretes = new PElement< Arete<S,T> >( nouvelleArete, lAretes);
debut->degre++; fin->degre++;

return nouvelleArete;
```

```
}
```

4.5 Nombre de sommets

```
template <class S, class T>
int Graphe<S,T>::nombreSommets() const
{
return PElement< Sommet<T> >::taille(lSommets);
}
```

4.6 Nombre d'arêtes

```
template <class S, class T>
int Graphe<S,T>::nombreAretes() const
{
return PElement< Arete<S,T> >::taille(lAretes);
}
```

4.7 Conversion en string et opérateur << d'écriture sur un flux

```
template <class S, class T>
Graphe<S,T>::operator string() const
{
ostringstream oss;
oss << "Graphe( \n";
oss << "prochaine clef = "<< this->prochaineClef << endl;
oss <<"nombre de sommets = "<< this->nombreSommets()<<"\n";

oss << PElement<Sommet<T> >::toString( lSommets, "", "\n", "");

oss <<"nombre d'arêtes = " << this->nombreAretes()<<"\n";

oss << PElement<Arete<S,T> >::toString( lAretes, "", "\n", "");
oss <<")";
return oss.str();
}

template <class S, class T>
ostream & operator << (ostream & os, const Graphe<S,T> & gr)
{
return os << (string)gr;
}
```

4.8 Methode *getAreteParSommets*

```
/**
```

```
 * cherche l'arête s1 - s2 ou l'arête s2 - s1 si elle existe
 *
 * DONNEES : s1 et s2 deux sommets quelconques du graphe
 * RESULTATS : l'arête s'appuyant sur s1 et s2 si elle existe, NULL
sinon
 *
 * */
template <class S, class T>
Arete<S,T> * Graphe<S,T>::getAreteParSommets( const Sommet<T> * s1,
const Sommet<T> * s2) const
{
PElement<Arete<S,T> > * l;

for ( l = this->lAretes; l; l = l->s)
    if ( l->v->estEgal(s1,s2))
       return l->v;

return NULL;
}
```

4.9 Liste d'adjacences d'un sommet

```
/**
recherche la liste des paires (voisin, arête) adjacentes de sommet
dans le graphe
*/
template <class S, class T>
PElement< pair< Sommet<T> *, Arete<S,T>* > >  *
Graphe<S,T>::adjacences(const Sommet<T> * sommet) const
{
const PElement< Arete<S,T> > * l;

PElement< pair< Sommet<T> *, Arete<S,T>* > > * r;
     // pair< Sommet<T> *, Arete<S,T>* >

for ( l = lAretes, r = NULL; l; l = l->s)

     if ( sommet == l->v->debut)
          r = new PElement< pair< Sommet<T> *, Arete<S,T>* > >
          ( new pair< Sommet<T> *, Arete<S,T>* >(l->v->fin,l->v),r);
     else
          if ( sommet == l->v->fin)
             r = new PElement< pair< Sommet<T> *, Arete<S,T>* > >
             ( new pair< Sommet<T> *, Arete<S,T>* >
               (l->v->debut,l->v),r);
return r;
}
```

5. Tests des classes GElement, Sommet, Arete et Graphe :

```
int main()
{
GElement e(5);

cout << "e = " << e << endl<< endl;

Sommet<double> s1(13,3.1416);
Sommet<string> s2(25,"aubergine");

cout << "s1 = " << s1 << endl;
cout << "s2 = " << s2 << endl<< endl;
```

```
Sommet<double> * debut = new Sommet<double>(23,2.71);
Sommet<double> * fin = new Sommet<double>(47,1.41);

Arete<string,double> a1(18, debut, fin, "melon");

cout << "a1 = " << a1 << endl<<endl;

Graphe<string, char> g;
Sommet<char> * so1 = g.creeSommet('A');
Sommet<char> * so2 = g.creeSommet('B');
Sommet<char> * so3 = g.creeSommet('C');

g.creeArete(so1,so2,"myrtille");
g.creeArete(so2,so3,"champignon");

cout << "g = " << g << endl;


char ch ; cin >> ch;
return 0;
}
```