# Exercice 40 - Graphes et STL

Un graphe dirigé peut être défini comme un couple $G = (V, E)$ où $V$ est un ensemble de *sommets* et $E$ un ensemble de couples $(i, j) \in V \times V$ que l'on appelle *arcs*. Un sommet $j \in V$ est dit *successeur* d'un sommet $i \in V$ si $(i, j) \in E$. Un sommet $j \in V$ est dit *prédécesseur* d'un sommet $i \in V$ si $(j, i) \in E$.

**Remarque :** en anglais, "sommet" se dit "vertex" (pluriel : "vertices"), et "arc" se dit "edge".

**Exemple :**

- $V = \{0, 1, 2, 3, 4, 5, 6\}$.
- $E = \{(2, 0), (2, 1), (2, 3), (3, 1), (1, 4), (6, 0), (6, 5), (6, 3), (0, 4), (4, 0), (4, 5), (4, 6)\}$.
- L'ensemble des successeurs du sommet 2 est $\{0, 1, 3\}$. L'ensemble des prédecesseurs de 4 est $\{0, 1\}$.

Pour manipuler un graphe dans un programme, une des structures de données les plus utilisées est la liste d'adjacence. Elle consiste en un tableau dont l'entrée $i$ donne la liste des successeurs du sommet $i$.

Le but de cet exercice est de se familiariser avec la STL en implémentant une classe de graphe simple puis une classe de graphe que l'on pourra paramétrer avec n'importe quel type pour représenter des sommets.

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

```
#if !defined(_GRAPH_H)
#define _GRAPH_H
#include<string>
#include<stdexcept>
using namespace std;
class GraphException : public exception {
 string info;
public:
 GraphException(const string& i) throw():info(i){}
 virtual ~GraphException() throw(){}
 const char* what() const throw(){ return info.c_str(); }
};
#endif
```

graph.h

Définir la fonction principale `main` dans le fichier `main.cpp`. S'assurer que le projet compile correctement.

### *Question 1*

Implémentez une classe graphe qui utilise un objet `vector<list<unsigned int> >` pour représenter la liste d'adjacence d'un graphe dont les $n$ sommets sont identifiés par les nombres de $\{0, 1, ..., n-1\}$. Implémentez toutes les méthodes de l'interface suivante :

```
class Graph {
 vector<list<unsigned int> > adj;
 string name;
public:
 Graph(const string& n, unsigned int nb);
 const string& getName() const;
 unsigned int getNbVertices() const;
 unsigned int getNbEdges() const;
 void addEdge(unsigned int i, unsigned int j);
 void removeEdge(unsigned int i, unsigned int j);
 const list<unsigned int>& getSuccessors(unsigned int i) const;
 const list<unsigned int> getPredecessors(unsigned int i) const;
};
ostream& operator<<(ostream& f, const Graph& G);
```

Le constructeur de la classe `Graph` prend en argument le nom du graphe ainsi que le nombre de sommets le constituant (qui n'évoluera pas au cours de la vie du graphe). Exploitez au mieux les algorithmes de la STL.

### *Question 2*

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 :

```
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 3* - Exercice d'approfondissement

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 4* - Exercice d'approfondissement

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{
  Graph G1("G1",5); cout<<G1;
  G1.addEdge(0,1); G1.addEdge(0,2); G1.addEdge(1,2); G1.addEdge(1,3);
  G1.addEdge(1,4); G1.addEdge(3,0);
  cout<<G1;
  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 G1 (5 vertices and 0 edges)
0:
1:
2:
3:
4:
graph G1 (5 vertices and 6 edges)
0:1 2
1:2 3 4
2:
3:0
4:
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
```