TP n°5

PRESENTATION DE L'APPLICATION

PRESENTATION DE L'ETUDE

Le but est de simuler le fonctionnement d'un réseau de Pétri. L'algorithmique d'évolution d'un réseau doit s'exécuter automatiquement en pas à pas. Pour simplifier, l'interface avec l'utilisateur se fera uniquement en mode texte.

Un réseau est constitué de places contenant un certain nombre de jetons ou marqueurs. Les jetons circulent le long des arcs fléchés en franchissant des transitions. Il est caractérisé par le nombre de places, le nombre de transitions, la liste des arcs reliant chaque transition aux places situées en amont et en aval. Enfin il évolue selon un critère mathématique à partir d'un marquage initial.

Un réseau est décrit par ses deux matrices, une d'entrée et une de sortie associées au vecteur de marquage initial. La matrice d'entrée représente en colonne la liste des places situées en amont de chaque transition. La matrice de sortie représente en colonne la liste des places situées en aval de chaque transition. Le marquage représente le nombre de marqueurs présents dans chaque place.

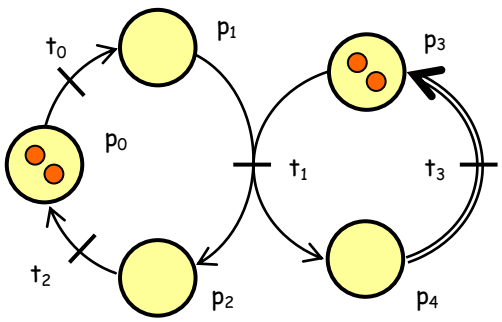
Le réseau peut se représenter dans un fichier de texte qui prend la forme suivante dans le cas de l'exemple ci-dessous, comportant cinq places, quatre transitions et quatre marqueurs. On constate que les deux arcs liés à la dernière transition sont doubles. Les places sont doublement présentes en entrée et en sortie.

Pour l'instant, on ne vous demande pas de comprendre le fonctionnement d'un réseau de Pétri, mais simplement sa représentation tabulaire illustrée ci-dessous.

```
E = "(5 4)"
" 1 0 0 0"
" 0 1 0 0"
" 0 0 1 0"
" 0 1 0 0"
" 0 0 0 2"

S = "(5 4)"
" 0 0 1 0"
" 1 0 0 0"
" 0 1 0 0"
" 0 0 0 2"
" 0 1 0 0"

M0 = "(5) 2 0 0 2 0";
```



EVOLUTION

Le marquage initial représente l'état du réseau avant le lancement. A chaque itération le marquage évolue vers le pas suivant en fonction du marquage actuel et des deux matrices. Le marquage suivant est obtenu en appliquant bêtement les formules indiquées ci-dessous. Pour chaque transition validée, on retire un marqueur par place amont et on ajoute un marqueur par place aval.

marquage = [p0, p1, p2, p3, p4]

validation = [t0, t1, t2, t3]

tj = sur tout i : { marquage(i) >= entree(i, j) }

$marquage^T = marquage^T - entree * validation^T + sortie * validation^T$

ORGANISATION DU TRAVAIL

EVALUATION

Le projet consiste à construire progressivement les objets nécessaires à cette application. A chaque étape de la construction du programme vous devez faire valider votre travail. La séquence de test et le résultat escompté sont donnés. C'est à vous de prévoir, pour chaque séquence de test, la ligne d'affichage des objets sur la console standard.

```
séquence de test ;
   // ==> affichage attendu
```

RECUPERATION DES ERREURS

Des erreurs sont détectées automatiquement par la bibliothèque standard. Certaines doivent être détectées au niveau local en lançant une exception personnalisée par exemple avec un texte.

```
if (condition) throw ("texte libre");
```

Le programme de test doit récupérer les erreurs éventuelles de la façon suivante.

```
try {
  // séquence à tester
  }
catch(const char * erreur) {
  std::cerr << erreur << std::endl;
  }
```

En fin de programme on récupère toutes les exceptions standards.

```
try {
  // corps du programme
  }
catch(const std::exception & erreur) {
  std::cerr << erreur.what() << std::endl;
  }
```

MARQUAGE

marquage (i) = nombre de marqueurs dans la place

DESCRIPTION

Le marquage est un vecteur d'entiers qui contient la liste des marqueurs présents pour chaque place.

```
#include <vector>

class Marquage {
  public:
    Marquage();
    inline const int get_places() const;
    void ajuster(int places);
    bool operator ==(const Marquage & right) const;
    int & operator ()(int place);
    int operator ()(int place) const;
    Marquage operator +(const Marquage & right) const;
    Marquage operator -(const Marquage & right) const;
    friend std::ostream & operator <<(std::ostream & str, const Marquage & right);
    friend std::istream & operator >>(std::istream & str, Marquage & right);
  protected:
    std::vector<int> marqueurs;
    int places;
};
```

CONSTRUCTION

Par défaut un marquage est vide. On peut ajuster le nombre de places après sa création. On utilise pour cela la méthode *resize()* du vecteur.

ACCES A UNE VALEUR

Chaque place est accessible en lecture ou en écriture par surcharge de l'opérateur de parenthèses. On utilise la méthode sécurisée *at()*du vecteur.

AFFICHAGE

Un marquage peut s'afficher dans un fichier de texte sous la forme d'une série de valeurs entières précédée par son nombre de places entre parenthèses. Un exemple est donné pour un marquage de cinq places.

```
(5) 0 1 2 3 4
```

TEST

```
Marquage m1;
  // ==> m1 = (0)
Marquage m2;
m2.ajuster(5);
  // ==> m2 = (5) 0 0 0 0 0
```

```
m2(3) = 8;
m2(2) = m2(3) - 3;
  // ==> m2 = (5) 0 0 5 8 0
```

VALIDATION 1.1

REMPLISSAGE

On peut remplir un marquage depuis un flux d'entrée standard contenant une donnée formatée par l'opérateur d'affichage précédent. On utilise la donnée entre parenthèses pour retailler le vecteur. Puis on itère dans le flux pour remplir le vecteur avec les valeurs suivantes. On admet que la donnée lue respecte le formatage attendu. Le flux peut contenir d'autres informations à la suite de la partie utile pour le marquage.

SUITE DU TEST

```
#include <sstream>

std::stringstream ss1;
ss1 << "(5) 8 7 6 5 4  (3) 3 2 1" << std::ends;
ss1 >> m1 >> m2;
  // ==> m1 = (5) 8 7 6 5 4
  // ==> m2 = (3) 3 2 1
```

VALIDATION 1.2

OPERATIONS

Deux opérateurs permettent d'additionner ou de soustraire des marquages. Ces opérations sont effectuées place par place. Si les marquages n'ont pas la même dimension, le résultat est ajusté au plus grand nombre de places. Les places manquantes sont traitées comme des zéros. Utiliser au maximum les propriété standard des vecteurs.

SUITE DU TEST

```
Marquage m3;
m3 = m1 + m2
  // ==> m3 = (5) 11 9 7 5 4
m3 = m2 - m1
  // ==> m3 = (5) -5 -5 -5 -5 -4
```

VALIDATION 1.3

MATRICE

matrice (i, j) = nombre d'arcs entre la place i et la transition j

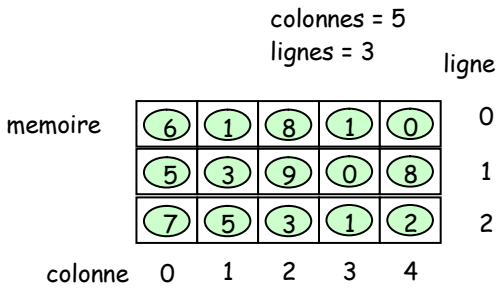
DESCRIPTION

Une matrice est une représentation tabulaire du nombre d'arcs associant une place à une transition. Elle est caractérisée par son nombre de places en ligne et son nombre de transitions en colonne. Ses dimensions sont nulles à la construction et modifiables par la suite.

Chaque élément est accessible en lecture ou en écriture par sa place et sa transition. La matrice est affichable dans un fichier de texte et inversement une série de valeurs dans un fichier de texte peut servir à l'initialiser.

```
class Matrice {
  public:
    Matrice();
    inline const int get_places() const;
    inline const int get_transitions() const;
    void ajuster(int places, int transitions);
    int & operator ()(int place, int transition);
    int operator ()(int place, int transition) const;
    friend std::ostream & operator <<(std::ostream & stream, const Matrice & right);
    friend std::istream & operator >>(std::istream & stream, Matrice & right);
  protected:
    int clef(int place, int transition) const;
    std::vector<int> arcs;
    int places;
    int transitions;
};
```

IMPLEMENTATION



ACCES A UNE VALEUR

La matrice est enregistrée en mémoire sous la forme d'un seul vecteur d'entiers. Les lignes sont enregistrées les unes après les autres de gauche à droite puis de haut en bas. La formule suivante donne l'index dans le vecteur pour une ligne et une colonne. L'accès à la clef doit être sécurisée et envoyer une exception en cas d'erreur sur le numéro de ligne ou de colonne..

```
clef = transition + place * transitions
```

SERIALISATION

Une matrice peut s'afficher dans un fichier de texte sous la forme d'une série de valeurs entières précédée par son nombre de lignes et de colonnes entre parenthèses. La représentation est tabulaire. Un exemple est donné pour une matrice de cinq lignes et quatre colonnes.

```
(5 4) 4 5 1 2  6 7 3 4  8 9 5 6  7 8 9 1  2 3 4 5
```

On peut modifier le contenu d'une matrice à partir d'un fichier de texte déjà rempli par l'opérateur d'affichage. La taille de la matrice est ajustée à celle du texte. La syntaxe du texte lu est supposée rigoureusement conforme au standard d'affichage.

TEST

```
Matrice x1;
  // ==> x1 = (0 0)
x1.ajuster(2, 3);
  // ==> x1 = (2 3)  0 0 0  0 0 0
std::stringstream ss2;
ss2 << "(5 4) 4 5 1 2  6 7 3 4  8 9 5 6  7 8 9 1  2 3 4 5";
ss2 >> x1;
  // ==> x1 = (5 4)  4 5 1 2  6 7 3 4  8 9 5 6  7 8 9 1  2 3 4 5
x1(5, 2) = x1(2, 1);
  // ==> erreur = place inconnue
x1(1, 4) = x1(2, 1);
  // ==> erreur = transition inconnue
x1(1, 2) = x1(2, 1);
  // ==> x1 = (5 4)  4 5 1 2  6 7 9 4  8 9 5 6  7 8 9 1  2 3 4 5
```

VALIDATION 2

VALIDATION

validation(j) = { sur tout i : marquage(i) >= entree(i, j) }

$marquage^T = marquage^T - entree * validation^T + sortie * validation^T$

DESCRIPTION

Un vecteur de validation booléen indique quelles sont les transitions franchissables. Ce vecteur de validation est construit à partir du vecteur de marquage et de la matrice d'entrée. La matrice doit avoir le même nombre de places que le marquage. Chaque élément du vecteur de validation représente une transition. Cette transition est franchissable si le vecteur de marquage est, pour toutes les places, supérieur à la colonne correspondante de la matrice d'entrée. Ceci signifie qu'il y a au moins un marqueur présent par place amont.

Le produit mixte d'une matrice par un vecteur de validation produit un vecteur de marquage temporaire ayant pour taille la hauteur de la matrice. La matrice doit avoir pour largeur la taille de la validation. Le résultat est un produit matriciel entre la matrice et un vecteur considéré comme une matrice à une colonne dans laquelle vrai est représenté par 1 et faux par 0. C'est à dire que pour chaque ligne de la matrice on calcule la somme des produits intermédiaires entre les valeurs ayant la même position.

```
class Validation {
  public:
    void valider(const Marquage & marquage, const Matrice & entree);
    Marquage marquer(const Matrice & matrice);
    friend std::ostream & operator <<(std::ostream & str, const Marquage & right);
  protected:
    std::vector<bool> transitions;
```

```
};
```

TEST

```
Validation vx;
  // ==> vx = (0)
std::stringstream ss3;
ss3 << "(5) 2 0 0 2 0";
ss3 << "(5 4) 1 0 0 0  0 1 0 0  0 0 1 0  0 1 0 0  0 0 0 2";
ss3 << "(5 4) 0 0 1 0  1 0 0 0  0 1 0 0  0 0 0 2  0 1 0 0";
Marquage m0;
Matrice entree;
Matrice sortie;
ss3 >> m0 >> entree >> sortie;
vx.valider(m0, entree);
  // ==> m0 = (5) 2 0 0 2 0
  // ==> vx = (4) 1 0 0 0
m0 = m0 - vx.marquer(entree) + vx.marquer(sortie);
  // ==> m0 = (5) 1 1 0 2 0
```

VALIDATION 3

RESEAU DE PETRI

DESCRIPTION

L'utilisateur charge à partir d'un flux en lecture la représentation du réseau comportant ses deux matrices d'entrée et de sortie suivies du marquage initial. La simulation s'exécute automatiquement pour un nombre d'itérations fixé à l'avance. Pour chaque transition validée, on retire un marqueur par place amont et on ajoute un marqueur par place aval.

```
class Reseau {
  public:
    Reseau();
    inline const Marquage get_marquage() const;
    inline const Matrice get_entree() const;
    inline const Matrice get_sortie() const;
    void iterer();
    friend std::ostream & operator <<(std::ostream & stream, const Reseau & right);
    friend std::istream & operator >>(std::istream & stream, Reseau & right);
  protected:
    Marquage marquage;
    Matrice entree;
    Matrice sortie;
    Validation validation;
};
```

TEST

```
Reseau rx;
std::stringstream ss3;
ss3 << "(5 4) 1 0 0 0  0 1 0 0  0 0 1 0  0 1 0 0  0 0 0 2";
ss3 << "(5 4) 0 0 1 0  1 0 0 0  0 1 0 0  0 0 0 2  0 1 0 0";
ss3 << "(5) 2 0 0 2 0";
```

```
ss3 >> rx;
  // ==> rx.get_marquage() = (5) 2 0 0 2 0
rx.iterer();
  // ==> rx.get_marquage() = (5) 1 1 0 2 0
rx.iterer();
  // ==> rx.get_marquage() = (5) 0 1 1 1 1
rx.iterer();
  // ==> rx.get_marquage() = (5) 1 0 1 0 2
rx.iterer();
  // ==> rx.get_marquage() = (5) 1 1 0 2 0
```

VALIDATION 4

ÉVOLUTION AUTOMATIQUE

Il faut itérer automatiquement sur un réseau jusqu'à la rencontre d'un marquage déjà rencontré. Une limite est fixée pour signaler que le réseau a peu de chance de converger.

L'utilisateur charge à partir d'un flux la représentation du réseau comportant les deux matrices d'entrée et de sortie suivies du marquage initial. Il associe ce réseau à l'historique puis lance la simulation qui s'exécute automatiquement jusqu'à convergence ou pour un nombre maximal d'itérations fixé.

```
class Historique {
  public:
    Historique();
    inline const Reseau * get_reseau() const;
    void set_reseau(Reseau * value);
    void lancer(int limite);
    friend std::ostream & operator<<(std::ostream & stream, const Historique & right);
  protected:
    Reseau * reseau;
    std::vector<Marquage> marquages;
};
```

TEST

```
Historique hx;
hx.lancer(20);
  // ==> erreur = pas de reseau associe
Reseau ry;
std::stringstream ss5;
ss5 << "(5 4) 1 0 0 0  0 1 0 0  0 0 1 0  0 1 0 0  0 0 0 2";
ss5 << "(5 4) 0 0 1 0  1 0 0 0  0 1 0 0  0 0 0 2  0 1 0 0";
ss5 << "(5) 2 0 0 2 0";
ss5 >> ry;
hx.set_reseau(&ry);
hx.lancer(20);
  // ==> hx = (5)
  // ==> pas 0 : marquage = (5) 2 0 0 2 0
  // ==> pas 1 : marquage = (5) 1 1 0 2 0
  // ==> pas 2 : marquage = (5) 0 1 1 1 1
  // ==> pas 3 : marquage = (5) 1 0 1 0 2
  // ==> pas 4 : marquage = (5) 1 1 0 2 0
```

VALIDATION 5

TP n° 2 : Jeux de Ballons

Introduction :

Le but de ce TP est de créer un projet `STADE` avec plusieurs classes (`Ballon, Joueur, But`) et de voir l'association d'une instance d'une classe avec plusieurs instance d'une autre classe. Le deuxième objectif est de manipuler certains itérateurs et conteneurs de la STL. Nous considérons les hypothèses suivantes :

- un joueur peut posséder le ballon qui peut être possédé par plusieurs joueurs (une mêlée par exemple),
- un ballon peut être dans un but qui contient éventuellement un ballon,
- un joueur peut marquer entre 0 et plusieurs buts,
- un but est marqué par un joueur exactement.

Une documentation sur la librairie STL peut être consultée à l'adresse suivante :
http ://www.sgi.com/tech/stl/

Préliminaires :

Le but de cette partie est d'écrire un programme qui saisit des informations au clavier et les affiche à l'écran.

Ecrire le contenu suivant dans le fichier stade.cpp :

```
// stade.cpp
#include <iostream>
void main ()
{
   cout << "Projet stade" << endl;
}
```

Pour saisir des informations au clavier, C++ dispose de l'opérateur >>. Pour manipuler des chaînes de caractères, C++ dispose de la classe string. Modifier stade.cpp comme suit :

```
#include <iostream>
#include <string>
void main ()
```

```
{
   cout << "Projet stade" << endl;
   string s;
   int i;
   cout << " Taper une chaine :" << endl;
   cin >> s;
   cout << " Taper un entier :" << endl;
   cin >> i;
   cout << "La chaine lue est : <" << s << ">" << endl;
   cout << "Le double de l'entier lu est :" << 2*i << endl;
}
```

Maintenant vous savez faire un programme qui lit une chaîne de caractères ou un entier sur l'entrée standard et qui affiche une chaîne de caractères sur la sortie standard.

Nous allons maintenant écrire un programme qui comprend 2 classes : une classe utilisatrice, la classe `Stade`, et une classe utilisée, la classe `Joueur`. La classe utilisatrice lit des informations sur les propriétés d'une instance à créer ; elle crée l'instance avec ses propriétés ; elle imprime l'instance sur la sortie standard. En plus du constructeur/destructeur, la classe `Joueur` est dotée de trois méthodes membre supplémentaires et deux attributs :

```
int age;
string nom;
Joueur (string , int );
void lireNom (string &);
void lireAge (int &);
void imprimer();
```

Les deux premières méthodes ont pour tâches de récupérer un nom et un âge tapés au clavier par l'utilisateur et de le donner en sortie (via la référence). Ne pas confondre ces deux méthodes avec les méthodes de lecture d'un attribut privé, ici tous les attributs et toutes le méthodes sont publiques.
Afin de tester la classe `Joueur`, modifier le programme principal comme suit :

```
#include <iostream>
#include <string>
#include "joueur.h"
void main ()
{
   cout << "Projet stade" << endl;
   string s;
   Joueur : :lireNom(s);
   int a;
   Joueur : :lireAge(a);
   cout << "s = <" << s << ">" « endl;
   cout << "a = <" << a << ">" << endl;
   Joueur * j = new Joueur(s, a);
   cout << "Joueur cree a l'adresse " << j << endl;
```

```
   j->imprimer();
}
```

Remarquer que le programme demande d'abord le nom et l'âge du joueur à l'utilisateur, puis connaissant les valeurs du nom et de l'âge, il crée une instance en appelant le constructeur de la classe `Joueur` avec ces valeurs. Remarquer que lors de l'appel de `lireNom` et `lireAge`, aucune instance de joueur n'existe. Elles sont des méthodes de classe et pas des méthodes d'instance. Maintenant vous savez donc faire un programme qui lit des propriétés d'une instance d'une classe, qui crée l'instance et qui l'affiche. Afin d'avoir une classe utilisatrice et une classe utilisée comme spécifié, déclarer une classe `Stade` avec une unique méthode de classe dans un fichier `stade.h`. Dans `stade.cpp`, le `main()` appelle cette méthode statique de la classe Stade.

Partie I :

Ici, nous remplaçons un joueur par des ballons. Nous allons créer une classe `Ballon` avec laquelle l'utilisateur pourra non seulement créer des ballons identifiés par une `string`, mais aussi les afficher et les détruire. Il va donc être nécessaire d'avoir un menu demandant l'action a effectuer (créer, détruire ou afficher) et un conteneur contenant les instances créées par l'utilisateur.

Ecrire un programme qui affiche le menu ci-dessous, prend la commande de l'utilisateur au clavier, affiche un message « choix n » (n=1,2,3,0) et recommence tant que l'utilisateur ne veut pas quitter le programme.

1. **créer un ballon**
2. **détruire un ballon**
3. **afficher la liste des ballons**
0. **quitter**

Enrichir le programme précédent en remplaçant l'affichage « choix 1 » par l'appel à la fonction membre `creer_une_instance()` de la classe `Ballon`. Pour cela, on pourra avoir une interface de la classe `Ballon` du type suivant :

```
class Ballon {
   public :
      string identificateur; // pour identifier un ballon
      int taille; // taille du ballon
      Ballon(string, int); // constructeur
      static void lireIdentificateur(string &); // dialogue
      static void lireTaille(int &); // dialogue
      void imprimer(); // imprimer
      static Ballon * creer_une_instance(); // dialogue puis new
};
```

`identificateur` est un attribut permettant d'identifier de manière unique une instance de la classe `Ballon`.

`taille` est un attribut descriptif d'une instance de la classe `Ballon`.

`lireIdentificateur(string &)` est une fonction qui demande à l'utilisateur de rentrer un identificateur au clavier, qui le récupère avec `cin` et qui le met dans le paramètre de type `string &`.

`lireAge(int &)` est une fonction qui demande à l'utilisateur de rentrer un `age` au clavier, qui le récupère avec `cin` et qui le met dans le paramètre de type `int &`. Ces deux dernières fonctions sont donc très simples. Pour cette raison, on peut choisir de ne pas les mettre sous forme de fonctions. Elles doivent être exécutées avant d'appeler le constructeur.

`Ballon(string, int)` est le constructeur d'une instance de la classe `Ballon`. On suppose que l'on connaît l'identificateur et la taille du ballon avant de l'appeler. On passe en paramètres l'identificateur et la taille du ballon à créer.

`imprimer()` est la fonction d'affichage à l'écran d'une instance. Par exemple, pour une instance dont l'identificateur vaut « bienGonflé » et dont la taille vaut 4, `imprimer()` peut afficher par exemple :

```
identificateur = bienGonflé
taille = 4
```

`Ballon * creer_une_instance()` est la fonction qui exécute les appels à `lireIdentificateur(string &)`, `lireAge(int &)`, `Ballon(string, int)` et `imprimer()`. Elle est appelée lorsque l'utilisateur tape 1 dans le menu `Ballon`. Elle fait exactement ce que faisait le programme principal de la partie précédente.

Enrichir le programme précédent en remplaçant l'affichage « choix 3 » par l'affichage de l'ensemble des instances de la classe `Ballon`. Pour cela, on enrichit la classe `Ballon` avec le vecteur des instances de la classe instances et une méthode d'impression de toutes les instances de la classe `imprimer_instances()` :

```
class Ballon {
  public :
    static vector<Ballon*> * instances; // les instances
    static void imprimer_instances(); // imprimer les instances
};
```

Le type `vector<Ballon*>` est un type paramétré. Il déclare un tableau unidimensionnel contenant des `Ballon*`. Sa taille s'adapte dynamiquement au nombre d'éléments du vecteur. La classe `vector<Ballon*>` est définie dans `vector.h` de la STL de C++.

`instances` est donc un pointeur vers un vecteur de `Ballon*`. `instances` sert à stocker les instances de la classe `Ballon`. Le mot-clé `static` est nécessaire pour préciser que cet attribut n'est pas un attribut d'instance mais un attribut de classe.

Pour utiliser correctement `instances`, il est nécessaire de veiller à plusieurs choses :

- il est déclaré avec `static` dans `ballon.h`, définir instances dans `ballon.cpp` :
  ```
  vector<Ballon*> * Ballon : :instances;
  ```

- son type est un pointeur, initialiser instances au début du `main()` :
  ```
  Ballon : :instances = new vector<Ballon*>();
  ```

- lors de la construction de l'instance, l'insérer dans le conteneur `instances` (l'insertion est faite au début par choix) :
  ```
  instances->insert(instances->begin(), this);
  ```

- lors de sa destrcution, enlever l'instance de `instances` :
  ```
  for (vector<Ballon*> : :iterator b=instances->begin();
  b!=instances->end(); b++)
  if (this==*b) { instances->erase(b); break; }
  ```

- inclure la définition de la classe `vector` partout où elle est utilisée :
  ```
  #include <vector>
  ```

Les méthodes présentées ci-dessous, sont définies implicitement par la classe template `vector` de la STL. `begin` retourne le premier itérateur du vecteur, `end` retourne le dernier.

```
vector<Ballon*> : :iterator begin(),
vector<Ballon*> : :iterator end(),
insert(vector<Ballon*> : :iterator, Ballon*),
erase(vector<Ballon*> : :iterator)
```

Un itérateur est une variable permettant de parcourir le vecteur ou de désigner une position dans le vecteur. Ici, son type est `vector<Ballon*> : :iterator`. Remarquer que la variable de la boucle for ci-dessus parcourant le vecteur est de type `vector<Ballon*> : :iterator.`

`insert` permet d'insérer un élément à une position (au début dans l'exemple ci-dessus) et `erase` permet d'enlever un élément du vecteur connaissant sa position.

Enrichir le programme précédent en remplaçant l'affichage « choix 2 » par l'appel à la fonction membre `detruire_une_instance()` de la classe `Ballon`. Pour cela, on pourra ajouter les membres suivants à la classe `Ballon` :

```
class Ballon {
  public :
     ~Ballon(); // destructeur
     static Ballon * getInstance(string); // retourne le ballon
     void toString(string &); // l'instance en string
     static bool detruire_une_instance(); // dialogue puis detruire
};
```

`toString(string &)` est la fonction qui met le descriptif d'un objet (identificateur + taille) dans une chaine de caractères de sorte que le contenu de `imprimer()` se réduise à :

```
     string s; toString(s); cout << s;
```

Par ailleurs, la classe string permet d'effectuer les opérations suivantes de manière agréable :

```
string s1 = "A l'impossible, ";
string s2 = s1;
string s3 = s2 + "nul n'est tenu.";
```

`Ballon * getInstance(string)` est la fonction qui retourne l'instance de la classe `Ballon` possédant un identificateur égal à la string passée en paramètre. Elle est appelée à chaque fois que l'on cherche une instance connaissant son identificateur. Elle retourne `NULL` si aucune instance ne correspond à la `string`. Elle retourne l'instance adéquate sinon. Pour écrire son contenu, on utilisera une boucle du style :

```
for (vector<Ballon*> : :iterator b=instances->begin();
   b!=instances->end(); b++)
      if ((*b)->identificateur==s) return *b;
return NULL;
```

`bool detruire_une_instance()` est la fonction qui exécute les appels à `lireIdentificateur(string&)`, `getInstance(string)`, `~Ballon()` et retourne `true` si la destruction est faite et `false` sinon. Elle est appelée lorsque l'utilisateur tape 2 dans le menu Ballon.

Partie II :

Dans cette partie, nous allons permettre en plus à l'utilisateur d'associer et dissocier des instances appartenant à des classes différentes. Ecrire un programme qui gère plusieurs classes, par exemple la classe `Joueur`, la classe `Ballon` et la classe `But`. En particulier, il affiche le menu ci-dessous :

1. **Joueur**
2. **Ballon**
3. **But**
0. **Quitter**

Puis, par exemple, dans le cas où l'utilisateur tape 2, le menu suivant apparait :

1. **créer un ballon**
2. **détruire un ballon**
3. **afficher la liste des ballons**
0. **précédent**

Enrichir le programme précédent en rajoutant la ligne suivante dans les menus `Ballon` et `But` :

4. associer un ballon à un but.

En particulier, on ajoutera un attribut et deux méthodes à la classe Ballon :

```
But * monBut;
static void associer_but_instance() {} // dialogue puis associer
void associer_but(But * b) {} // associer ballon a but
```

Analogue pour la classe But :

```
Ballon * monBallon;
static void associer_ballon_instance() {} // dialogue puis associer
void associer_ballon(Ballon * b) {} // associer but a ballon
```

Enrichir le programme précédent en rajoutant la ligne suivante dans le menu `Ballon` :

5. associer un ballon à des joueurs.

Et la ligne suivante dans le menu `Joueur` :

5. associer un joueur à un ballon.

En particulier, on ajoutera un attribut `mesJoueurs` à la classe `Ballon` :

```
vector<Joueur*> * mesJoueurs;
```

Et un attribut à la classe `Joueur` : `Ballon * monBallon;`

Remarquer qu'il est nécessaire d'utiliser un attribut de type `vector<Joueur*>` dans la classe `Ballon` pour associer des joueurs et un ballon. Par contre, cela n'est pas nécessaire dans l'autre sens.

Enrichir le programme précédent en rajoutant la ligne suivante dans le menu `Ballon` **:**

6. dissocier un ballon d'un joueur.

Et la ligne suivante dans le menu `Joueur` :

6. dissocier un joueur d'un ballon.

Compléter votre programme pour que tout lien puisse être dissocié.

Partie III :

Pour visualiser les instances du programme, il est nécessaire d'avoir deux méthodes qui transforment une instance en `string`. Une première méthode qui transforme une instance en une `string` identifiant l'instance (nous l'appellerons `toIdent()`) et une seconde méthode qui transforme l'instance en une `string` contenant tous les noms des attributs de l'instance avec leur valeurs.

Ecrire la méthode `toString()` et la méthode `toIdent()` pour toutes les classes de votre application. Par exemple, pour la classe `Ballon`, le corps de ces méthodes est :

```
void Ballon : :toString(string & s) {
   char ss[16];
   sprintf (ss, "%d", taille);
   s = "Identificateur " + identificateur + "\n Taille " + ss + "\n
But ";
   if (monBut) s = s + monBut->identificateur;
   else s = s + "null ";
   s = s + "\nJoueurs : ";
   string sss = "";
   for (Joueur ** j = mesJoueurs->begin(); j != mesJoueurs->end();
j++) {
      ((Joueur*)*j)->toIdent(sss);
   }
   s = s + sss + "\n";
}

void Ballon : :toIdent(string & s) {
   s = s + identificateur + " ";
}
```