# Laboratoire de Programmation en C++

**2ème informatique et systèmes :**
**option(s) industrielle et réseaux (1er et 2ème quart)**
**et 2ème informatique de gestion (1er et 2ème quart)**

**Année académique 2016-2017**

# *Gestion de clubs de Tenni de Table*

**Anne Léonard**
**Denys Mercenier**
**Claude Vilvens**
**Jean-Marc Wagner**

# 0. Introduction

## 0.1 Informations générales : UE, AA et règles d'évaluation

Cet énoncé de laboratoire concerne les unités d'enseignement (UE) suivantes :

### a) 2ème Bach. en informatique de Gestion : « Développement Système et orienté objet »

Cette UE comporte les activités d'apprentissage (AA) suivantes :

- **Base de la programmation orientée objet – C++** (45h, Pond. 30/90)
- **Principes fondamentaux des Systèmes d'exploitation** (15h, Pond. 10/90)
- **Système d'exploitation et programmation système UNIX** (75h, Pond. 50/90)

Ce laboratoire intervient dans la construction de la côte de l'AA « Base de la programmation orientée objet – C++ ».

### b) 2ème Bach. en informatique et systèmes : « Développement Système et orienté objet »

Cette UE comporte les activités d'apprentissage (AA) suivantes :

- **Base de la programmation orientée objet – C++** (45h, Pond. 45/101)
- **Système d'exploitation et programmation système UNIX** (56h, Pond. 56/101)

Ce laboratoire intervient dans la construction de la côte de l'AA « Base de la programmation orientée objet – C++ ».

Quelque soit le bachelier, la cote de l'AA « Base de la programmation orientée objet – C++ » est construite de la même manière :

♦ théorie : un examen écrit en janvier 2017 (sur base d'une liste de questions fournies en novembre et à préparer) et coté sur 20;
♦ laboratoire **(cet énoncé)** : 2 évaluations (aux dates précisées dans l'énoncé de laboratoire), chacune cotée sur 20; la moyenne arithmétique pondérée (30% pour la première partie et 70% pour la seconde partie) de ces 2 cotes fournit une note de laboratoire sur 20;
♦ note finale : **moyenne arithmétique de la note de théorie (50%) et de la note de laboratoire (50%)**.

Cette procédure est d'application tant en 1ère qu'en 2ème session.

**1)** Chacun des membres d'une équipe d'étudiants doit être capable d'expliquer et de justifier l'intégralité du travail (pas seulement les parties du travail sur lesquelles il aurait plus particulièrement travaillé)

**2)** En 2ème session, un **report de note** est possible pour chacune des deux notes de laboratoire ainsi que pour la note de théorie **pour des notes supérieures ou égales à 10/20**.

Toutes les évaluations (théorie ou laboratoire) ayant des **notes inférieures à 10/20** sont **à représenter dans leur intégralité**.

**3)** Les consignes de présentation des dossiers de laboratoire sont fournies par les différents professeurs de laboratoire via leur centre de ressources

## 0.2 Le contexte : la gestion de clubs de Tennis de Table

Les travaux de Programmation Orientée Objets (POO) C++ se déroulent dans le contexte de la gestion de clubs de Tennis de Table. Plus particulièrement, il s'agit de gérer les joueurs, leur classements et leurs performances ainsi que de former des équipes qui pourront se rencontrer. Les responsables de clubs ou de la fédération, les secrétaires, interviendront en tant qu'utilisateur de l'application finale.

En Belgique, il existe deux fédérations de Tennis de Table, la fédération royale et la fédération ouvrière (pour plus d'infos voir http://users.skynet.be/frottbfl/index.htm). C'est cette dernière qui a été considérée pour ce dossier de programmation. Afin de se concentrer sur l'essentiel (càd la programmation en C++), des simplifications dans la gestion des classements ont été réalisées.

## 0.3 Philosophie du laboratoire

Le laboratoire de programmation C++ sous Unix a pour but de vous permettre de faire concrètement vos premiers pas en C++ au 1er quart puis de conforter vos acquis au 2ème quart. Les objectifs sont au nombre de trois :

- mettre en pratique les notions vues au cours de théorie afin de les assimiler complètement;
- créer des "briques de bases" pour les utiliser ensuite dans une application de synthèse;
- vous aider à préparer l'examen de théorie du mois de janvier;

Le dossier est prévu à priori pour **une équipe de deux étudiants** qui devront donc se coordonner intelligemment et se faire confiance. Il est aussi possible de présenter le travail seul (les avantages et inconvénients d'un travail par deux s'échangent).

Il s'agit bien d'un laboratoire de C++ sous UNIX. La machine de développement sera Sunray. Même s'il n'est pas interdit (que du contraire) de travailler sur un environnement de votre choix (**Dev-C++** sur PC/Windows sera privilégié car compatible avec C++/Sunray – à la rigueur Visual C++ sous Windows, g++ sous Linux, etc …) à domicile, **seul le code compilable sous Sunray sera pris en compte !!!** Une machine virtuelle possédant exactement la même configuration que celle de Sunray sera mise à la disposition des étudiants lors des premières séances de laboratoire.

Un petit conseil : ***lisez bien l'ensemble de l'énoncé*** avant de concevoir (d'abord) ou de programmer (après) une seule ligne ;-). Dans la deuxième partie (au plus tard), prévoyez une schématisation des diverses classes (diagrammes de classes **UML**) et élaborez d'abord "sur papier" (donc sans programmer directement) les divers scénarios correspondant aux fonctionnalités demandées.

## 0.4 Méthodologie de développement

La programmation orientée objet permet une approche modulaire de la programmation. En effet, il est possible de scinder la conception d'une application en 2 phases :

1. La programmation des classes de base de l'application (les briques élémentaires) qui rendent un service propre mais limité et souvent indépendant des autres classes. Ces modules doivent respecter les contraintes imposées par « un chef de projet » qui sait comment ces classes vont interagir entre elles. Cette partie est donc réalisée par « le programmeur créateur de classes ».
2. La programmation de l'application elle-même. Il s'agit d'utiliser les classes développées précédemment pour concevoir l'application finale. Cette partie est donc réalisée par « le programmeur utilisateur des classes ».

Durant la première partie de ce laboratoire (**1er quart**), vous vous situez en tant que « programmeur créateur de classes ». On va donc vous fournir une série de 7 jeux de test (les fichiers Test1.cpp, Test2.cpp, …, Test7.cpp) qui contiennent une fonction main() et qui vous imposeront le comportement (l'interface) de vos classes.

Dans la deuxième partie du laboratoire (**2ème quart**), vous vous situerez en tant que « programmeur utilisateur des classes » utilisant les classes que vous aurez développées précédemment. C'est dans cette seconde phase que vous développerez l'application elle-même.

## 0.5 Planning et contenu des évaluations

**a) Evaluation 1 (continue) :**

**Porte sur** : les 5 premiers jeux de tests (voir tableau donné plus loin).

**Date de remise du dossier** : le 1er lundi du 2ème quart à 12h00 au plus tard.

**Modalités d'évaluation** : à partir du 1er lundi du 2ème quart selon les modalités indiquées par le professeur de laboratoire.

**b) Evaluation 2 (examen de janvier 2017) :**

**Porte sur** : les classes développées dans les jeux de tests 6 et 7, et le développement de l'application finale (voir tableau donné plus loin).

**Date de remise du dossier** : jour de votre examen de Laboratoire de C++ (selon horaire d'examens)

**Modalités d'évaluation** : selon les modalités fixées par le professeur de laboratoire.

**CONTRAINTES** : Tout au long du laboratoire de C++ (évaluation 1 et 2, et seconde session), il vous est interdit, pour des raisons pédagogiques, d'utiliser la classe **string** et les **containers génériques template de la STL**.

# 1. Première Partie : Création de diverses briques de base nécessaires (Jeux de tests)

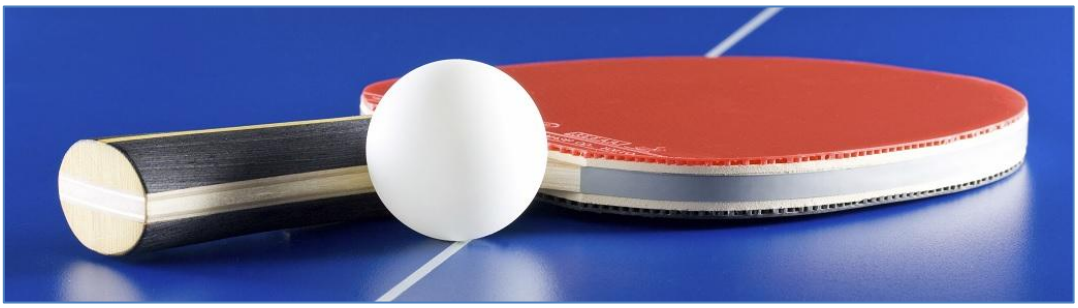
| Points principaux de l'évaluation | Subdivisions |
|---|---|
| **EVALUATION 1 (Jeux de tests 1 à 5) → 1er lundi du 4ème quart** | |
| • **Jeu de tests 1** : Implémentation d'une classe de base (**Joueur)** | Constructeurs, destructeurs |
| | Getters, Setters et méthode Affiche |
| | Fichiers Event.cpp, Event.h et makefile |
| • **Jeu de tests 2 :** Agrégation entre classes (classe **Joueur**) | Agrégation par valeur (classe **Matricule)** |
| | Agrégation par référence (classe **Classement**) |
| | Variables membres statiques |
| • **Jeu de tests 3 :** Surcharge des opérateurs | Operateurs = de la classe Joueur |
| | Opérateurs - de la classe Classement |
| | Opérateurs ==, < et > de Classement et Joueur |
| | Opérateurs << >> de Classement et << de Joueur |
| | Opérateurs ++ et – de la classe Classement |
| | Opérateurs ++, --, + et – de la classe Joueur |
| • **Jeu de tests 4 :** Héritage et virtualité | Classe de base **Personne** |
| | Classe abstraite dérivée **Membre** |
| | Classes dérivées **Joueur** et **Secretaire** |
| | Test des méthodes (non) virtuelles |
| • **Jeu de tests 5 :** Exceptions | **InvalidClassementException** |
| | **InvalidPasswordException** |
| | Utilisation correcte de **try**, **catch** et **throw** |
| **EVALUATION 2 (Jeux de tests 6 à 7) → Examen de Janvier 2017** | |
| • **Jeu de test 6 :** Containers génériques | Classe **abstraite ListeBase** |
| | Classe **Liste** (→ int et Classement) |
| | Classe **ListeTriee** (→ int et Classement) |
| | **Itérateur** : classe **Iterateur** |
| • **Jeu de test 7 :** Flux | Méthodes **Save()** et **Load()** de Classement et Matricule |
| | Méthodes **Save()** et **Load()** de Joueur |
| | Opérateurs << et >> de Classement |

# 1.1 Jeu de tests 1 (Test1.cpp) :

# Une première classe

## a) Description des fonctionnalités de la classe

Un des éléments principaux de l'application est évidemment la notion de joueur de tennis de table. En se limitant à l'essentiel, un joueur possède un nom, un prénom, ainsi qu'un numéro de club (La fédération assigne un numéro entier à chaque club). D'autres caractéristiques plus précises seront abordées plus loin.



Notre première classe, la classe **Joueur**, sera donc caractérisée par :

- Un **nom** : une chaine de caractères allouée dynamiquement (**char ***) en fonction du texte qui lui est associé.
- Un **prénom** : une chaine de caractères allouée dynamiquement (**char ***) en fonction du texte qui lui est associé.
- Un **numéro de club** : un entier (**int**) permettant d'identifier de manière unique un club affilié à la fédération.

Comme vous l'impose le premier jeu de test (Test1.cpp), on souhaite disposer au minimum des trois formes classiques de constructeurs et d'un destructeur, des méthodes classiques getXXX() et setXXX() et une méthode pour afficher les caractéristiques de l'objet. Les variables de type chaîne de caractères seront donc des char*. **Pour des raisons purement pédagogiques, le type string (de la STL) NE pourra PAS être utilisé du tout dans TOUT ce dossier de C++.** Vous aurez l'occasion d'utiliser la classe string dans votre apprentissage du C# et du Java.

## b) Méthodologie de développement

Veillez à tracer (cout << …) vos constructeurs et destructeurs pour que vous puissiez vous rendre compte de quelle méthode est appelée et quand elle est appelée.

On vous demande de créer pour la classe Joueur (ainsi que pour chaque classe qui suivra) les fichiers .cpp et .h et donc de travailler en fichiers séparés. Un makefile permettra d'automatiser la compilation de votre classe et de l'application de tests.

# 1.2 Jeu de tests 2 (Test2.cpp) : Associations entre classes : agrégations

Nous allons à présent ajouter à notre classe Joueur les informations concernant son identification auprès de la fédération (son matricule) mais également son classement qui traduit son niveau d'habileté dans ce sport. Pour cela, nous allons donc créer deux classes supplémentaires.

## a) La classe Matricule (Essai1())

Il s'agit à présent de créer une classe permettant d'identifier de manière unique un joueur au niveau de la fédération. On vous demande de créer la classe **Matricule** contenant

- Une variable **numero** de type **int** qui identifie de manière unique un joueur auprès de la fédération.
- Une variable **dateInscription** de type **char[11]** (chaine de caractères de taille fixe) qui contient la date d'inscription du joueur sous la forme « XX/XX/XXXX ». Exemple : « 25/09/2016 ».
- Un constructeur par défaut, un de copie et un d'initialisation (voir jeu de tests), ainsi qu'un destructeur (dans la suite, nous n'en parlerons plus → toute classe digne de ce nom doit au moins contenir un constructeur par défaut et un de copie, ainsi qu'un destructeur).
- Les méthodes getXXX()/setXXX() associées,
- Une méthode Affiche() permettant d'afficher les caractéristiques d'un matricule.

## b) La classe Classement (Essai2())

Tout joueur débutant n'a pas de classement, on dit qu'il est « non-classé » (NC). Dès qu'il a gagné suffisamment de points (voir plus loin), il obtient un premier classement qu'il pourra améliorer dans la suite en fonction de ses performances. Tout classement est composé d'une lettre et d'un nombre concaténés. L'échelle de classement (du plus fort au plus faible) est : A1, A2, A3, …, A10, B1, B2, …, B6, C1, C2, …,C6, D1, D2, …, D6, E1, E2, …, E6, F1 et F2. Il y a donc 6 lettres possibles, 10 nombres pour la lettre A, 2 pour la lettre F et 6 pour les autres lettres. Le plus petit classement que peut posséder un joueur est donc F2.

On vous demande donc de créer la classe **Classement** contenant

- Une variable **lettre** de type **char** qui contient la lettre du classement.
- Une variable **nombre** de type **short** qui contient le numéro du classement.
- Les différents constructeurs et un destructeur. Un objet classement construit par défaut sera initialisé à F2.
- La méthode **Affiche()** affichera le classement sous la forme « F2 », « B4 », …

### c) Agrégations à la classe Joueur (Essai3())

Tout joueur normalement inscrit possède donc un matricule et peut avoir un classement (ou être NC). Nous pouvons compléter la classe Joueur avec :

- Une variable **matricule** de type **Matricule**.
- Un pointeur **classement** de type **Classement*** pointant vers un objet de type Classement si le joueur possède un classement, ou valant NULL si le joueur est non-classé.
- Un 2ème constructeur d'initialisation (voir jeu de tests). Une classe peut avoir plusieurs constructeurs d'initialisation. Par défaut ou par initialisation, un objet joueur est construit sans classement (NULL).
- Les méthodes getXXX()/setXXX() associées aux deux nouvelles variables membres.
- La méthode Affiche() doit être adaptée pour tenir compte de ces 2 nouvelles variables.

Bien sûr, les classes **Matricule** et **Classement** doivent posséder leurs propres fichiers .cpp et .h.

La classe Joueur contenant une variable membre dont le type est une autre classe, on parle d'**agrégation par valeur**, l'objet de type Matricule fait partie intégrante de l'objet Joueur.

La classe Joueur possède à présent également un pointeur vers un objet de la classe Classement. Elle ne contient donc pas l'objet Classement en son sein mais seulement un pointeur vers un tel objet. On parle d'**agrégation par référence**.

### d) Mise en place de quelques variables statiques utiles (Essai4())

On se rend bien compte que de nombreux joueurs sont classés F2 et F1 étant donné le « faible » niveau de ces classements. Dès lors, on pourrait imaginer de créer des objets « permanents » (dits « statiques ») existant même si le programmeur de l'application n'instancie aucun objet et représentant le classement F1 (par exemple), voir le classement A1 (pour se faire plaisir ☺, on peut toujours rêver ☺ !!! Un seul joueur dans toute la fédération possède ce classement).

Dès lors, on vous demande d'ajouter, à la classe Classement, **2 variables membres, appelées F1 et A1, statiques constantes** de type **Classement** et représentant les classements F1 et A1. Voir jeu de tests.

# 1.3 Jeu de tests 3 (Test3.cpp) : Extension des classes existantes : surcharges des opérateurs

Il s'agit ici, de surcharger un certain nombre d'opérateurs des classes développées ci-dessus afin de faciliter la gestion des classements des joueurs, ainsi que de leurs points accumulés grâce à leurs performances. Avant cela, quelques précisions sur le fonctionnement des classements des joueurs :

- Au départ, un joueur dispose d'un nombre de points égal à zéro (qu'il s'agisse d'un NC ou d'un joueur qui vient d'acquérir un nouveau classement).
- Un joueur peut gagner (en gagnant des matchs) ou perdre (en perdant des matchs) des points.
- Un joueur qui bat un joueur de même classement que lui gagne 1 point, 2 points pour un joueur de classement directement supérieur à lui, 3 points pour un joueur de 2 classements supérieurs à lui, etc…
- Un joueur qui perd contre un joueur de même classement que lui perd 1 point, 2 points pour un joueur de classement directement inférieur à lui, 3 points pour un joueur de 2 classements inférieurs à lui, etc…
- Dès qu'un joueur a accumulé +20 points, il passe au classement directement supérieur et son total de points repasse à 0. Sauf le classement A1 (le plus haut) pour lequel les points continuent à s'accumuler.
- Dès qu'un joueur a accumulé -20 points, il passe au classement directement inférieur et son total de points repasse à 0. Sauf pour le classement F2 (le plus bas) pour lequel les points continuent à s'accumuler négativement. Tout joueur qui a acquis un classement ne pourra jamais redevenir NC.
- Un joueur NC dispose également d'un nombre de points, au départ égal à 0, mais qui ne peut jamais être négatif, même s'il perd beaucoup de matchs. Dès qu'il a accumulé 20 points, il passe F2 et son total de points repasse à 0.

Dès lors, avant tout chose, on vous demande d'ajouter à la classe **Joueur** :

- Une variable membre **points** de type **int** contenant le total de points accumulés par le joueur. Au départ, cette variable sera égale à 0. (Revoir les constructeurs !!!).
- Les méthodes getXXX()/setXXX() associées.

### a) Surcharge de l'opérateur = de la classe Joueur (Essai1())

Dans un premier temps, on vous demande de surcharger l'opérateur = de la classe Joueur, permettant d'exécuter un code du genre :

```
Joueur j,j1(…);
j1.setPoints(10) ;

j = j1 ;
```

### b) Surcharge de l'opérateur – de la classe Classement (Essai2())

Il s'agit à présent de mettre en place un opérateur permettant de calculer facilement les points gagnés/perdus lorsqu'un joueur gagne/perd un mach. On vous demande donc de surcharger l'opérateur – de la classe Classement retournant la différence de points entre deux classements :

```
Classement c1('A',7), c2("B4"), c3('B',4) ;
int diff = c1-c2 ;  // diff contient la valeur 7
diff = c2-c1 ;      // diff vaut à présent -7
diff = c2-c3 ;      // diff vaut à présent 0
```

### c) Surcharge des opérateurs de comparaison des classes Classement et Joueur (Essai3() et Essai4())

A terme, il sera nécessaire d'ordonner les joueurs par ordre de classement. Dans ce but, on vous demande de surcharger les opérateurs <, > et == des classes Classement et Joueur permettant d'exécuter un code du genre

```
Classement c1,c2;
…
if (c1 > c2) cout << "c1 est plus fort que c2" ;
if (c1 == c2) cout << autre message;
if (c1 < c2) …

Joueur j1,j2;
…
if (j1 > j2) cout << "j1 est plus fort que j2" ;
if (j1 == j2) cout << autre message;
if (j1 < j2) …
```

Les joueurs sont comparés uniquement sur leurs classements sans tenir compte de leurs points respectifs. En d'autres termes, deux joueurs ayant même classement mais des points différents sont supposés égaux.

Remarque : il serait judicieux d'utiliser l'opérateur – de Classement pour coder les opérateurs de comparaison de Classement ☺…

### d) Surcharge des opérateurs d'insertion << et d'extraction >> (Essai5())

On vous demande à présent de surcharger les operateurs << et >> de la classe Classement, ce qui permettra d'exécuter un code du genre :

```
Classement c1;
cout << "Classement :";
cin >> c1; //permet d'encoder un classement sous la forme d'une chaine de caractères "D3"
cout << c1 ; //affiche "D3"

Joueur j1 ;
…
cout << j1 ; // on se limitera aux données principales :
             // → « Wagner Jean-Marc (club=112, classement=D2, points=0) »
```

**e) Surcharge des opérateurs ++ et –- de classe Classement (Essai6() et Essai7())**

On vous demande de programmer les opérateurs de post et pré-in(dé)crémentation de la classe Classement. Ceux-ci in(dé)crémenteront un objet Classement **de un classement**. Cela permettra d'exécuter le code suivant :

```
Classement c1('E',4), c2("D1"), c3('B',5), c4("E6") ;

cout << ++c1 << endl ; // c1 vaut a présent "E3"
cout << c2++ << endl ; // c2 vaut à présent "C6"
cout << --c3 << endl ; // c3 vaut à présent "B6"
cout << c4-- << endl ; // c4 vaut à présent "F1"
```

**f) Surcharge des opérateurs ++ et –- de classe Joueur (Essai8() et Essai9())**

On vous demande de programmer les operateurs de post et pré-in(dé)crémentation de la classe Joueur. Ceux-ci in(dé)crémenteront la variable point d'un joueur **de une unité**. Attention ! Si la valeur de la variable points atteint +20 ou -20, le joueur change de classement ! Cela permettra d'exécuter le code suivant :

```
Joueur j1;
Classement c("E3");
j1.setClassement(&c) ;
j1.setPoints(18);
j1++ ; // j1 est toujours E3 et possède 19 points
j1++ ; // j1 est maintenant E2 et possède 0 points
++j1 ; // j1 est E2 et possède 1 points
j1-- ; // j1 est E2 et possède 0 points
--j1 ; // j1 est E2 et possède -1 points
```

**f) Surcharge des opérateurs + et - de classe Joueur (Essai10())**

On vous demande de programmer les operateurs + et – de la classe Joueur permettant de mettre à jour les points et le classement d'un joueur qui a gagné/perdu un certain nombre de points. Cela permettra d'exécuter le code suivant :

```
Joueur j1,j2;
Classement c("E3");
j1.setClassement(&c) ;
j1.setPoints(12);
j2 = j1 + 5;   // j1 reste inchangé, j2 est identique à j1 sauf qu'il possède 17 points
j2 = j2 + 8 ;  // j2 est maintenant E2 et possède 5 points
j2 = j1 - 15 ; // j1 reste inchangé, j2 est identique à j1 sauf qu'il possède -3 points
j2 = j2 - 20 ; // j2 est maintenant E4 et possède -3 points
```

Remarque : pour programmer ces deux opérateurs, il serait judicieux d'utiliser les opérateurs ++ et – de la classe Joueur ☺…

# 1.4 Jeu de tests 4 (Test4.cpp) :

# Associations de classes : héritage et virtualité

L'application finale fera intervenir les joueurs, qui ne pourront pas utiliser l'application, mais également les secrétaires qui pourront, par l'intermédiaire d'un couple login/password, utiliser l'application. Quelques considérations sont à savoir :

- Chaque club dispose d'un secrétaire responsable de la gestion des joueurs de son club.
- Chaque club est affilié à la fédération de Tennis de Table et y est identifié par son numéro de club.
- La fédération (qui gère les divisions, le planning des rencontres, etc…) est elle-même « dirigée » par un ensemble de secrétaires « fédéraux ».
- Tous les secrétaires (de club ou de fédération) pourront se connecter à l'application. Ils doivent donc disposer un couple login/password.

Nous remarquons que tous les intervenants de l'application ont au moins un point commun : un nom et un prénom. En effet, il s'agit tous de personnes ! De plus, tous les joueurs et les secrétaires sont membres d'une association, que ce soit un club (pour les joueurs et les secrétaires de club), ou la fédération (pour les secrétaires fédéraux). Ils disposent donc tous en plus d'un point commun, un nombre entier représentant le numéro du club ou 0 dans le cas des secrétaires fédéraux. Toutes ces considérations nous mène à concevoir la petite hiérarchie de classes décrite ci-dessous.

### a) Une classe de base : la classe Personne (Essai1())

L'idée est de concevoir une hiérarchie de classes, par héritage, dont la classe de base regroupera les caractéristiques communes de tous les individus. Cette classe de base sera la classe **Personne** ayant les variables membres suivantes :

- Un **nom** : une chaine de caractères allouée dynamiquement (**char ***) en fonction du texte qui lui est associé.
- Un **prenom** : une chaine de caractères allouée dynamiquement (**char ***) en fonction du texte qui lui est associé.

Elle reprend donc les caractéristiques de base d'un joueur, mais également d'un secrétaire. Comme toute classe digne de ce nom, elle doit disposer des constructeurs habituels, destructeur, accesseurs, d'une méthode Affiche() et au minimum des opérateurs <<, >> et = (voir jeu de tests).

### b) Une première classe héritée : la classe abstraite Membre

On vous demande de programmer la classe **abstraite Membre**, héritant de la classe Personne et qui possède :

- Un entier (**int**) **numClub** : représentant le numéro d'association (club/fédération)
- La **méthode virtuelle pure** **Affiche()** : ce qui assure que la classe Membre est ***abstraite***.

Cette classe modélise tout individu faisant partie d'une association, et numClub identifie cette association. Dans le cas d'un joueur ou d'un secrétaire de club, il s'agit du numéro du club (> 0). Dans le cas d'un secrétaire fédéral, numClub est égal à 0.

### c) Héritage : les classes dérivées de la hiérarchie (Essai2() et Essai3())

Maintenant que nous disposons des classes regroupant tous les points communs des intervenants de l'application, on vous demande programmer les classes dérivées décrites ci-dessous.

Déjà connue, mais revisitée, on vous demande de re-programmer la classe **Joueur**, qui hérite de la classe **Membre**, et qui présente, en plus, les variables membres suivantes :

- Une variable **matricule** de type **Matricule**.
- Un pointeur **classement** de type **Classement*** pointant vers un objet de type Classement.
- Une variable **points** de type **int** contenant les points de performance du joueur.
- Toutes les méthodes et opérateurs déjà mis en place précédemment.
- La méthode **Affiche()** qui affiche toutes les caractéristiques du joueur.

Un secrétaire est un membre d'une association qui a, en plus, un login et un password qui lui permettront de se connecter à l'application. Donc, on vous demande de programmer la classe **Secretaire**, qui hérite de la classe Membre, et qui présente, en plus, les variables membres suivantes :

- Un **login** : une chaîne de caractères (**char[9]**) de taille fixe qui contiendra un login de 8 caractères maximum.
- Un **password** : une chaîne de caractères (**char[9]**) de taille fixe qui contiendra un mot de passe de 8 caractères.
- Toutes les méthodes nécessaires habituelles (…)
- La méthode **Affiche()** qui affiche toutes les caractéristiques d'un secrétaire.

On redéfinira bien entendu les méthodes de la classe de base lorsque c'est nécessaire, par exemple les opérateurs d'affectation = ainsi que << et >>.

### d) Mise en évidence de la virtualité et du down-casting (Essai4() et Essai5())

La méthode **Affiche()** étant virtuelle, on vous demande de :

- comprendre et savoir expliquer le code de l'essai 4 mettant en évidence la virtualité de la méthode Affiche().
- comprendre et savoir expliquer le code de l'essai 5 mettant en évidence le down-casting et le dynamic-cast du C++.

# 1.5 Jeu de tests 5 (Test5.cpp) :

# Les exceptions

On demande de mettre en place une structure minimale de gestion des erreurs propres aux classes développées jusqu'ici. On va donc imaginer quelques classes d'exception du type suivant :

- **InvalidClassementException** : lancée lorsque l'on tente de créer ou modifier un objet de la classe Classement avec des données invalides, c'est-à-dire si le classement n'est pas formé par la concaténation d'une lettre valide et d'un nombre valide. Par exemple, si c est un objet de la classe Classement, c.setLettre('J') ou c.setClassement("F3") lancera une exception. Attention que c.setNombre(4) lancera également une exception si la lettre est F, de même que c.setLettre('E') si le nombre n'est pas inférieur ou égal à 6, etc... Cette exception est également lancée par les opérateurs ++ et -- si on tente d'incrémenter le classement A1 (qui est le plus haut) ou de décrémenter le classement F2 (qui est le plus bas). La classe **InvalidClassementException** contiendra une seule variable membre du type **chaine de caractères (char *)** qui contiendra un message lié à l'erreur, comme par exemple « Lettre invalide ! » ou « Classement mal formé ! ».

- **InvalidPasswordException** : lancée lorsque l'on tente d'affecter à un secrétaire un mot de passe qui ne respecte pas la condition suivante : « Un mot de passe valide contient exactement 8 caractères dont au moins une lettre et au moins un chiffre ». Par exemple, si s est un objet de la classe Secretaire, s.setPassword("azerty") lancera une exception. La classe **InvalidPasswordException** contiendra
  - une variable membre **code** de type **int** contenant le code de l'erreur. Ce code peut prendre une des 3 valeurs suivantes :
  - Trois **variables membres statiques constantes de type int** BAD_LENGTH_ERROR, MISSING_ALPHA_ERROR et MISSING_DIGIT_ERROR représentant les 3 types d'erreur qui peuvent survenir.
  - Une méthode **void perror(const char* message)** qui affiche le message passé en paramètre concaténé avec un message expliquant la cause de l'erreur (Par exemple « Probleme:Missing alpha error » si message contient « Probleme »).

Le fait d'insérer la gestion d'exceptions implique qu'elles soient récupérées et traitées lors des tests effectués en première partie d'année (**il faudra donc compléter le jeu de tests Test5.cpp** → utilisation de **try**, **catch** et **throw**), mais également dans l'application finale.

# 1.6 Jeu de tests 6 (Test6.cpp) :

# Les containers et les templates

## a) L'utilisation future des containers

On conçoit sans peine qu'une application de gestion de clubs de Tennis de Table va utiliser des containers mémoire divers qui permettront par exemple de contenir tous les joueurs ou toutes les équipes d'un club. Nous allons ici mettre en place une base pour nos containers. Ceux-ci seront construits via une hiérarchie de classes templates.

## b) Le container typique : la liste

Le cœur de notre hiérarchie va être une liste chaînée dynamique. Pour rappel, une liste chaînée dynamique présente un pointeur de tête et une succession de cellules liées entre elles par des pointeurs, la dernière cellule pointant vers NULL. Cette liste va être encapsulée dans une **classe abstraite ListeBase template** contenant comme seule variable membre le pointeur de tête de la liste chaînée. Elle aura donc la structure de base suivante :

```
template<class T> class ListeBase
{
	Protected :
		Cellule<T> *pTete ;
	…
}
```

où les cellules de la liste chaînée auront la structure suivante :

```
template<class T> struct Cellule
{
	T valeur ;
	Cellule<T> *suivant ;
}
```

La classe **ListeBase** devra disposer des méthodes suivantes :

- Un **constructeur par défaut** permettant d'initialiser le pointeur de tête à NULL.
- Un **constructeur de copie**.
- Un **destructeur** permettant de libérer correctement la mémoire.
- La méthode **estVide()** retournant le booléen true si la liste est vide et false sinon.
- La méthode **getNombreElements()** retournant le nombre d'éléments présents dans la liste.
- La méthode **Affiche()** permettant de parcourir la liste et d'afficher chaque élément de celle-ci.

- La **méthode virtuelle pure** **T* insere(const T & val)** qui permettra, une fois redéfinie dans une classe héritée, d'insérer un nouvel élément dans la liste, à un endroit dépendant du genre de liste héritée (simple liste, pile, file, liste triée, …) et de retourner l'adresse de l'objet T inséré dans la liste. **Attention !!!** Le fait que la méthode insere retourne un pointeur (non constant) permettra de modifier l'objet T pointé. Une modification d'une des variables membres de cet objet sur laquelle portent les opérateurs de comparaison <, >, ou == pourrait endommager une liste triée (qui ne le serait donc plus ☹). A utiliser avec prudence !
- Un **opérateur =** permettant de réaliser l'opération « liste1 = liste2 ; » sans altérer la liste2 et de telle sorte que si la liste1 est modifiée, la liste2 ne l'est pas et réciproquement.

### c) Une première classe dérivée : La liste simple

Nous disposons à présent de la classe de base de notre hiérarchie. La prochaine étape consiste à créer la **classe template Liste** qui hérite de la classe ListeBase et qui redéfinit la méthode insere de telle sorte que **l'élément ajouté à la liste soit inséré à la fin de celle-ci**.

Dans un premier temps, vous testerez votre classe Liste avec des **entiers**, puis ensuite avec des objets de la classe **Classement**.

Bien sûr, on travaillera, comme d'habitude, en fichiers séparés afin de maîtriser le problème de l'instanciation des templates.

### d) La liste triée

On vous demande à présent de programmer la **classe template ListeTriee** qui hérite de classe ListeBase et qui redéfinit la méthode insere de telle sorte que l'élément ajouté à la liste soit inséré au bon endroit dans la liste, c'est-à-dire en respectant l'ordre défini par les opérateurs de comparaison de la classe template.

Dans un premier temps, vous testerez votre classe ListeTriee avec des **entiers**, puis ensuite avec des objets de la classe **Classement**. Ceux-ci devront bien sûr être triés par ordre de force.

## e) Parcourir et modifier une liste : l'itérateur de liste

Dans l'état actuel des choses, nous pouvons ajouter des éléments à une liste ou à une liste triée mais nous n'avons aucun moyen de parcourir cette liste, élément par élément, afin d'en modifier un et encore moins d'en supprimer un. La notion d'itérateur va nous permettre de réaliser ces opérations.

On vous demande donc de créer la classe **Iterateur** qui sera un **itérateur** de la classe **ListeBase** (elle permettra donc de parcourir tout objet instanciant la classe Liste ou ListeTriee), et qui comporte, au minimum, les méthodes et opérateurs suivants:

- **reset()** qui réinitialise l'itérateur au début de la liste.
- **end()** qui retourne le booléen true si l'itérateur est situé au bout de la liste.
- **Opérateur ++** qui déplace l'itérateur vers la droite.
- **Opérateur de casting ()** qui retourne (par valeur) l'élément pointé par l'itérateur.
- **Opérateur &** qui retourne l'adresse de l'élément (objet T) pointé par l'itérateur.
- **remove()** qui retire de la liste et retourne l'élément pointé par l'itérateur.

L'application finale fera un usage abondant de la classe ListeTriee. On vous demande donc d'utiliser la classe Iterateur afin de vous faciliter l'accès aux containers. Son usage sera vérifié lors de l'évaluation finale.

# 1.7 Jeu de tests 7 (Test7.cpp)

# Première utilisation des flux

Il s'agit ici d'une première utilisation des flux en distinguant les flux caractères (manipulés avec les opérateurs << et >>) et **les flux bytes (méthodes write et read**).

**a) La classe Joueur se sérialise elle-même (Essai1, Essai2 et Essai3)**

On demande de compléter la classe **Joueur** avec les deux méthodes suivantes :

- **Save(ofstream & fichier) const** permettant d'enregistrer sur flux fichier toutes les données d'un joueur (nom, prénom, numéro de club, matricule, classement et points) et cela champ par champ. Le fichier obtenu sera un fichier **binaire** (utilisation des méthodes **write** et **read**).

- **Load(ifstream & fichier)** permettant de charger toutes les données relatives à un joueur enregistré sur le flux fichier passé en paramètre.

Afin de vous aider dans le développement, on vous demander d'utiliser l'encapsulation, c'est-à-dire de laisser chaque classe gérer sa propre sérialisation. En d'autres termes, on vous demande d'ajouter aux classes **Personne**, **Membre**, **Classement** et **Matricule** les méthodes suivantes :

- **void Save(ofstream & fichier) const** : méthode permettant à un objet de s'écrire lui-même sur le flux fichier qu'il a reçu en paramètre.
- **void Load(ifstream & fichier)** : méthode permettant à un objet de se lire lui-même sur le flux fichier qu'il a reçu en paramètre.

Ces méthodes seront appelées par les méthodes Save et Load de la classe Joueur lorsqu'elle devra enregistrer ou lire ses variables membres dont le type n'est pas un type de base.

Tous les enregistrements seront de taille variable. Pour l'enregistrement d'une chaine de caractères « chaine » (type **char ***), on enregistrera tout d'abord le nombre de caractères de la chaine (strlen(chaine)) puis ensuite la chaine elle-même. Ainsi, lors de la lecture dans le fichier, on lit tout d'abord la taille de la chaine et on sait directement combien de caractères il faut lire ensuite.

**b) Création, écriture et lecture d'un premier fichier texte (Essai4)**

On demande de créer un petit fichier texte contenant quelques classements. Pour ce faire, vous devez utiliser les opérateurs << et >> de la classe Classement. Normalement, si ceux-ci ont été bien programmés, vous n'avez rien à faire ☺ !

# 2. Deuxième Partie : Développement de l'application

To be continued ☺…