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 2015-2016

Gestion d'un horaire de cours

Anne Léonard
Denys Mercenier
Claude Vilvens
Jean-Marc Wagner

1. Introduction

1.1 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 2016 (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

1.2 Le contexte : la gestion d'un horaire de cours

Les travaux de Programmation Orientée Objets (POO) se déroulent dans le contexte de la gestion d'un horaire de cours. Plus particulièrement, il s'agit de gérer la planification de cours, ainsi que d'éviter les conflits entre les ressources tels que les professeurs, les groupes d'étudiants et les locaux, principaux intervenants dans l'application.

Dans un premier temps, il s'agira de modéliser la gestion d'un horaire en général, ce qui fera intervenir la notion d'événement (Quoi ? Où ? Quand ?) et de timing (Quel jour ? Quelle heure ? Combien de temps ?) Nous aborderons ensuite, la problématique de la gestion des « planifiables » (Qui peut avoir un horaire ?), et des cours donnés par les professeurs à un ou plusieurs groupes en même temps (cours théorique/laboratoire).

L'application finale sera utilisée par toute personne réalisant la confection d'un horaire de cours. Elle lui permettra

- De concevoir un horaire de cours sur une semaine (horaire qui se répète donc de semaine en semaine),
- D'ajouter des professeurs, locaux, groupes à un horaire (ajout manuel ou importation),
- De planifier des cours pour ces différents intervenants en évitant les conflits,
- D'exporter l'horaire pour n'importer quel intervenant.

1.3 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.

1.4 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.cxx, Test2.cxx, …, Test7.cxx) 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.

1.5 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 2016) :

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**.

2. 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 (**Event**)	Constructeurs, destructeurs
	Getters, Setters et méthode Affiche
	Fichiers Event.cxx, Event.h et makefile
• **Jeu de tests 2 :** Agrégation entre classes (classe **Event**)	Agrégation par valeur (classe **Temps**)
	Agrégation par référence (classe **Timing**)
	Variables membres statiques
• **Jeu de tests 3 :** Surcharge des opérateurs	Operateurs = + - de la classe Temps
	Opérateurs < > == de la classe Temps
	Opérateurs << et >> de la classe Temps
	Opérateurs ++ et -- de la classe Temps
	Opérateurs < > == de la classe Timing
• **Jeu de tests 4 :** Héritage et virtualité	Classe abstraite de base **Planifiable**
	Classes dérivées **Professeur, Groupe et Local**
	Test des méthodes (non) virtuelles
• **Jeu de tests 5 :** Exceptions	**InvalidTempsException**
	InvalidJourException
	Utilisation correcte de **try**, **catch** et **throw**
EVALUATION 2 (Jeux de tests 6 à 7) → Examen de Janvier 2016	
• **Jeu de test 6 :** Containers génériques	Classe **abstraite ListeBase**
	Classe **Liste** (→ int et Timing)
	Classe **ListeTriee** (→ int et Timing)
	Itérateur : classe **Iterateur**
• **Jeu de test 7 :** Flux	Méthodes **Save()** et **Load()** de Heure et Timing
	Méthodes **Save()** et **Load()** de Event

2.1 Jeu de tests 1 (Test1.cxx) :

Une première classe

a) Description des fonctionnalités de la classe

Un des éléments de base de l'application est la notion d'événement (« Event »). Un événement est « quelque chose » qui se produit un certain jour, à une certaine heure, à un certain endroit et qui dure un certain temps. Dans cette première partie, nous allons commencer par aborder le « quoi ? » et le « ou ? ». La notion de « Quand ? » sera abordée plus loin.

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

- Un **code** : un entier (**int**) permettant d'identifier de manière unique un événement dans l'horaire.
- Un **intitulé** : une chaine de caractères allouée dynamiquement (**char ***) en fonction du texte qui lui est associé. Cet intitulé représente le « quoi » d'un événement. Exemples : « Cinéma avec les potos », « Resto avec David », « Labo C++ », …
- Un **lieu** : une chaîne de caractères de taille fixe (**char [21]**) représentant l'endroit où se passe l'événement.

Comme vous l'impose le premier jeu de test (Test1.cxx), 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* ou 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 Event (ainsi que pour chaque classe qui suivra) les fichiers .cxx 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.

2.2 Jeu de tests 2 (Test2.cxx) : Associations entre classes : agrégations

Nous allons à présent ajouter à notre classe Event tout ce qui est nécessaire à la gestion du temps, c'est-à-dire « Quel jour ? », « A quelle heure ? » et « Combien de temps ? ».

a) Une agrégation par valeur (Essai1() et Essai2())

Il s'agit à présent de créer une classe modélisant la notion d'heure de la journée/de durée. On vous demande de créer la classe **Temps** contenant

- Une variable **heure** de type **int** contenant l'heure de la journée/le nombre d'heures entières de la durée. Exemple : pour « 15h39 », heure contient 15.
- Une variable **minute** de type **int** contenant les minutes. Exemple : pour « 15h39 », minute contient 39.
- Un constructeur par défaut, un de copie et deux 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 l'heure/la durée sous la forme XXhXX.

Ensuite, on vous demande de créer une classe **Timing** contenant toutes les informations temporelles d'un événement :

- Un **jour** : une chaine de caractères allouée dynamiquement (**char ***) en fonction du texte qui lui est associé, et contenant « Lundi » ou « Mardi », …
- Une **heure** (variable de type **Temps**) représentant l'heure du début de l'événement.
- Une **durée** (variable de type **Temps**) représentant la durée de l'événement.

De nouveau, cette classe doit contenir les méthodes getXXX()/setXXX() associées à ses variables membres (ce que nous ne répéterons plus par la suite pour les autres classes), ainsi qu'une méthode Affiche().

Bien sûr, les classes Temps et Timing doivent posséder leurs propres fichiers .cxx et .h. La classe Timing contenant des variables membres dont le type est une autre classe, on parle d'**agrégation par valeur**, les objets de type Heure font partie intégrante de l'objet Timing.

b) Une agrégation par référence (Essai3())

Nous pouvons à présent compléter la classe Event en ajoutant

- la variable **timing** (du type **Timing***) : il s'agit d'un pointeur vers un objet du type Timing et représentant donc toute l'information temporelle de l'événement,
- les méthodes setTiming(…) et getTiming().

Notez que la méthode Affiche() de la classe Event doit être mise à jour pour tenir compte du timing de l'événement.

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

c) Mise en place de quelques variables statiques utiles (Essai4())

On se rend bien compte que la variable jour de classe Timing ne peut contenir que des chaînes de caractères bien précises, à savoir les jours de la semaine. Dès lors, on vous demande d'ajouter, à la classe Timing, **7 variables membres statiques constantes** de type chaîne de caractères (**char ***) : **LUNDI** contenant « Lundi », **MARDI** contenant « Mardi », etc…

Afin d'assurer l'unicité de chaque événement (càd de la variable code de la classe Event), nous allons devoir gérer un indice courant « global » qui sera utilisé et incrémenté chaque fois que l'on créera un nouvel événement. Pour cela, on vous demande d'ajouter, à la classe Event, une **variable membre statique** **codeCourant** de type **int**. Cette variable sera initialisée à 1 et incrémentée au besoin.

2.3 Jeu de tests 3 (Test3.cxx) : 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 heures et durées.

a) Surcharge des opérateurs =, + et – de la classe Temps (Essai1(), Essai2() et Essai3())

Dans un premier temps, on vous demande de surcharger les opérateurs =, + et - de la classe Heure, permettant d'exécuter un code du genre :

```
Temps h1,h2(13,30),h3,h4,d(30);

h1 = h2 ;       // h1 vaut à présent 13h30

h3 = h2 + 20 ;  // attention, h2 doit rester inchangé, h3 vaut 13h50
h3 = h3 + 30 ;  // h3 vaut à présent 14h20
h3.Affiche() ;
h4 = 10 + h3 ;  // h4 vaut à présent 14h30
h4 = h4 + d ;   // h4 vaut à présent 15h00

h3 = h3 - 10 ;             // h3 vaut à présent 14h10
Temps duree = 50 - d ;     // duree vaut à présent 0h20
Temps duree2 = h3 - h1 ;   // duree2 vaut 0h40
```

On supposera que le résultat d'une opération d'addition ou de soustraction donne un résultat compris entre 0h00 et 23h59. Les dépassements ne sont pas acceptés et seront gérés plus tard.

b) Surcharge des opérateurs de comparaison de la classe Heure (Essai4())

A terme, il sera nécessaire de classer les événements par ordre chronologique. Dans ce but, on vous demande de surcharger les opérateurs <, > et == de la classe Heure, permettant d'exécuter un code du genre

```
Heure h1,h2 ;
…
if (h1 > h2) cout << "h1 est posterieure h2" ;
if (h1 == h2) cout << autre message;
if (h1 < h2) …
```

c) Surcharge des opérateurs d'insertion et d'extraction de la classe Temps (Essai5())

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

```
Temps h1,h2;

cin >> h1 ; // permet d'encoder une heure en tapant « 14h20 »
cin >> h2 ;
cout << "De" << h1 << "a" << h2 << endl ; // Affiche l'heure sous forme XXhXX
```

d) Surcharge des opérateurs ++ et – de classe Temps (Essai6())

On vous demande donc de programmer les operateurs de post et pré-in(dé)crémentation de la classe Temps. Ceux-ci in(dé)crémenteront l'heure/la durée de **30 minutes**. Cela permettra d'exécuter le code suivant :

```
Temps h1(13,10), h2(16,50), h3(16,0), h4(9,50) ;

cout << ++h1 << endl ; // h1 vaut a présent 13h40
cout << h2++ << endl ; // h2 vaut à présent 17h20
cout << --h3 << endl ; // h3 vaut à présent 15h30
cout << h4-- << endl ; // h4 vaut à présent 9h20
```

e) Surcharge des opérateurs de comparaison de la classe Timing (Essai7())

Comme cela a été fait pour la classe Temps, on vous demande de surcharger les opérateurs <, > et == de la classe Timing afin d'assurer l'ordre chronologique. Ceci permettra d'exécuter un code du genre

```
Timing t1,t2 ;
…
if (t1 > t2) cout << "t1 est posterieur que t2" ;
if (t1 == t2) cout << autre message;
if (t1 < t2) …
```

La comparaison porte tout d'abord sur le jour de la semaine. A jour égal, on compare l'heure de début. A jour et heure identiques, on dira que l'événement le plus court (en durée) est antérieur à l'autre.

2.4 Jeu de tests 4 (Test4.cxx) :

Associations de classes : héritage et virtualité

Nous allons à présent aborder la modélisation des intervenants du type « Qui peut avoir un horaire ? Qui est occupé pendant cette tranche horaire ? etc… » Dans notre application de gestion d'horaires de cours, trois types d'intervenant existent :

- Les professeurs, qui ont un nom, un prénom et un « horaire », c'est-à-dire une liste de cours à donner.
- Les groupes d'étudiants qui ont un numéro (ex : 2201, 2222, 2225,…) et un « horaire », c'est-à-dire une liste de cours à suivre.
- Les locaux qui ont un nom (ex : AN, PV11, LE0, …), un nombre de places et un « horaire », c'est-à-dire une liste des cours ayant lieu dans chaque local.

Nous remarquons que tous ces intervenants ont au moins un point commun : un horaire, c'est-à-dire une liste de cours (type d'événement particulier). Nous donnerons à tous ces intervenants, pouvant avoir un « horaire », « être planifiés », l'appellation de « planifiable ». Ces planifiables se différencient par un type : « Sont-ils des individus ? Ou sont-ils une ressource matérielle utilisée par ces individus ? ».

a) Héritage : une classe Abstraite de base

Etant donné les considérations évoquées ci-dessus, l'idée est de concevoir une hiérarchie de classes, par héritage, dont la classe de base regroupera les caractéristiques communes aux trois intervenants, à savoir une liste de cours, mais également un type. Cette classe de base sera la **classe abstraite Planifiable** et contiendra, pour l'instant, la variable membre suivante :

- Un **type** : un simple caractère (**char**) caractérisant le type de planifiable. Exemples : « h » pour « humain », « m » pour « matériel », etc…

La liste des cours sera gérée plus tard, dans la seconde partie du laboratoire, ainsi que la classe Cours qui n'est rien d'autre qu'un Event spécialisé (plus tard donc ☺).

Afin de vous faire comprendre la notion de virtualité et donc d'être capable de différencier les méthodes virtuelles et les méthodes non virtuelles, la classe Planifiable contiendra les méthodes :

- **void Affiche()** : qui affiche la seule caractéristique connue actuellement, c'est-à-dire « Planifiable humain » ou « Planifiable matériel ». **Cette fonction doit être non-virtuelle, et ce, pour des raisons purement pédagogiques**.
- **char* getIdentifiant()** : qui retourne une chaine de caractères identifiant un planifiable (cette chaine sera construite de manière dynamique en fonction des données propres à chaque planifiable ; voir plus bas) et **qui doit être une méthode virtuelle pure**. Cette fonction n'a donc pas de corps et devra obligatoirement être redéfinie dans les classes dérivées.

b) Héritage : les classes dérivées de la hiérarchie

Maintenant que nous disposons de la classe mère de la hiérarchie, on vous demande programmer les classes dérivées décrites ci-dessous.

Un professeur est un planifiable qui a, en plus, un nom et un prénom. On vous demande donc de programmer la classe **Professeur**, qui hérite de la classe Planifiable, et qui présente, en plus, les variables membres suivantes :

- Un **nom** : chaine de caractères (**char ***),
- Un **prenom** : chaine de caractères (**char ***).

Un groupe d'étudiants est un planifiable qui a, en plus, un numéro. Donc, on vous demande de programmer la classe **Groupe**, qui hérite de la classe Planifiable, et qui présente, en plus, la variable membre suivante :

- Un **numero** : un entier (**int**).

Enfin, un local est un planifiable qui a, en plus, un nom et un nombre de places. Vous devez donc programmer la classe **Local**, qui hérite de la classe Planifiable, et qui présente, en plus, les variables membres suivantes :

- Un **nom** : chaine de caractères (**char ***).
- La variable **nbPlaces** : un entier (**int**).

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

c) Mise en évidence de la virtualité

Les trois classes dérivées (Professeur, Groupe, Local) devront redéfinir les méthodes suivantes :

- **void Affiche()** : qui affiche à présent les caractéristiques propres de chaque intervenant. Pour rappel, la méthode Affiche() a été définie en tant que non-virtuelle dans la classe Planifiable.
- **char* getIdentifiant()** : qui crée et retourne une chaine de caractères décrite plus bas. Pour rappel, cette méthode a été déclarée en tant que virtuelle pure dans la classe Planifiable.

La chaine de caractères retournée par la méthode getIdentifiant() contiendra différentes informations en fonction des cas :

Professeur	→	« nom prenom» (Ex : « Wagner Jean-Marc »)
Groupe	→	« Gnumero » (Ex : « G2201 »)
Local	→	« nom(nbPlaces) » (Ex : « AE(80) »)

2.5 Jeu de tests 5 (Test5.cxx) :

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 :

- **InvalidTempsException** : lancée lorsque l'on tente de créer ou modifier un objet de la classe Temps avec des données invalides, c'est-à-dire si heure est inférieure à 0 ou supérieure à 23, et/ou si minute est inférieure à 0 ou supérieure à 59. Par exemple, si h est un objet de la classe Temps, h.setMinute(70) lancera une exception. Cette exception est également lancée par les opérateurs + et – si le résultat est antérieur à 0h00 ou postérieur à 23h59, mais aussi lors d'une extraction du flux en cas de mauvaise saisie (ex : « xh56 »). La classe **InvalidTempsException** contiendra une seule variable membre du type **chaine de caractères** (**char ***) et qui contiendra un message lié à l'erreur, comme par exemple « Nombre de minutes invalide ! ».

- **InvalidJourException** : lancée lorsque l'on tente d'affecter à un objet de la classe Timing un jour de la semaine qui ne fait pas partie des 7 variables membres statiques de la classe. Par exemple, si t est un objet de la classe Timing, t.setJour(« Monday ») lancera une exception. La classe **InvalidJourException** contiendra une simple **variable membre statique constante de type char*** contenant le message « Jour invalide » et la méthode getMessage() permettant de retourner ce message.

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.cxx** → utilisation de **try**, **catch** et **throw**), mais également dans l'application finale.

2.6 Jeu de tests 6 (Test6.cxx) :

Les containers et les templates

a) L'utilisation future des containers

On conçoit sans peine qu'une application de gestion d'horaire va utiliser des containers mémoire divers qui permettront par exemple de contenir tous les événements ou tous les planifiables entrant en jeu. 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** permettant d'initialiser le pointeur de tête à NULL.
- 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 !

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 au début de celle-ci**.

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

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 **Timing**. Ceux-ci devront bien sûr être triés par ordre chronologique.

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.

2.7 Jeu de tests 7 (Test7.cxx)

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**). Dans cette première utilisation, nous ne traiterons que des flux bytes.

La classe Event se sérialise elle-même

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

- **Save(ofstream & fichier) const** permettant d'enregistrer sur flux fichier toutes les données de l'événement (identifiant, intitulé, lieu, Timing) 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 événement 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 **Timing** et **Heure** 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 Event lorsqu'elle devra enregistrer ou lire sa variable membre de type Timing qui fera de même pour lire ses variables de type Heure.

Tous les enregistrements seront de taille variable. Pour l'enregistrement d'une chaine de caractère « 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.

3. Deuxième Partie : Développement de l'application

Points principaux de l'évaluation	subdivisions
EVALUATION 2 (Développement de l'application) → Examen de Janvier 2016	
• **Gestion des planifiables :**	Fonctionnalités respectées ?
	Instanciation et utilisation de **ListeTriee<Professeur>, ListeTriee<Groupe>, ListeTriee<Local>**
	Lecture et parsing des **fichiers textes** (.csv)
• **Gestion de l'horaire :**	Fonctionnalités respectées ?
	Classe **Cours** héritant de **Event**
	Liste des groupes dans la classe Cours : **Liste<int>**
	Instanciation et utilisation de **ListeTriee<Cours>**
	Classe **SmartPointer**
	ListeTriee<SmartPointer<Cours>> dans la classe Planifiable → horaire des planifiables
	Classe **Horaire** (BONUS) ?
	Exportation de l'horaire au **format texte** (.txt)
• **Menu Fichier :**	Fonctionnalités respectées ?
	Gestion d'un fichier binaire (*.hor)
• **Utilisation des briques de base**	Utilisation de l'itérateur **Iterateur**
	Utilisation de **InvalidTempsException** et de **InvalidJourException**
	Création et utilisation d'autres exceptions ?

3.1 L'application proprement dite et son menu

L'application vise essentiellement un seul profil d'utilisateur : un horairiste en charge de la confection d'un horaire de cours sur un site particulier. Dès lors, cette application sera installée à terme sur la machine d'un horairiste et une gestion d'entrée en session via un couple « login/mot de passe » n'est pas à envisager.

Les fichiers gérés par cette application seront de 3 types :

- Des **fichiers binaires** (ex : « 1erQuadrimestre_2015.hor ») qui contiendront toutes les informations d'un horaire (voir plus loin) de telle sorte qu'un horaire complet soit stocké dans un seul de ces fichiers,
- Des **fichiers csv** permettant d'importer des professeurs, groupes, locaux sans devoir les ajouter un par un manuellement dans l'application, et
- Des **fichiers textes** qui seront le format de sortie de l'horaire, et donc distribués aux professeurs, groupes, etc…

Une fois démarrée, l'application présentera un menu du type suivant :

```
**********************************************************************
********** Gestion d'un horaire de Cours ***************************
**********************************************************************
      0. Quitter
Fichier ***** En cours : 1erQuadrimestre_2015.hor ******************
      1. Nouvel Horaire
      2. Ouvrir un horaire
      3. Enregistrer l'horaire
      4. Fermer l'horaire
Gestion des Planifiables *****************************************
      5. Ajouter un Professeur/Groupe/Local
      6. Importer des Professeurs/Groupes/Locaux au format csv
      7. Afficher la liste des Professeurs/Groupes/Locaux
      8. Supprimer un Professeur/Groupe/Local
Gestion de l'horaire *********************************************
      9. Planifier un Cours
      10. Déplanifier un Cours
      11. Déplanifier tous les Cours d'un Professeur/Groupe/Local
      12. Afficher tous les Cours d'un Professeur/Groupe/Local
      13. Afficher tous les Cours d'un jour particulier
Publier l'horaire ************************************************
      14. Exporter l'horaire d'un Professeur/Groupe/local au format txt
```

Afin de travailler sur un horaire, il est nécessaire de créer un nouvel horaire/ouvrir un horaire existant. Le menu ci-dessus montre qu'actuellement le fichier « 1erQuadrimestre_2015.hor » est ouvert et en cours de gestion. Les différents items du menu seront expliqués plus loin. Nous allons tout d'abord mettre en place quelques classes supplémentaires et l'architecture de l'application.

3.2 Classes supplémentaires et architecture de l'application

a) Une classe spécialisée pour les Cours

Il s'agit tout d'abord de développer la classe Cours modélisant un cours donné par un professeur, dans un certain local, à un certain nombre de groupes, à un jour et une heure donnée. Il s'agit d'un événement spécialisé. On vous demande donc de développer la classe **Cours**, qui hérite de la classe Event, et qui présente les variables membres supplémentaires suivantes :

- Un **professeur** : chaine de caractères de type **char*** allouée dynamiquement et contenant l'identifiant d'un professeur (chaîne obtenue par la méthode getIdentifiant() d'un professeur),
- Une liste « **groupes** » : variable du type **Liste<int>** contenant le numéro de tous les groupes concernés par ce cours.

Héritant de la classe Event, la classe Cours dispose des variables membres

- **code** (entier qui représente de manière unique un cours donné)
- **intitulé** qui représente ici le nom du cours donné (Exemple : « Labo C++ »)
- **lieu** qui représente le local où sera donné le cours. Il s'agira de l'identifiant du Local (chaîne de caractères obtenue par la méthode getIdentifiant() d'un local)
- **timing** qui contient toute l'information temporelle du cours (jour, heure, durée)

La classe Cours devra être munie des opérateurs de comparaison de telle sorte que l'ordre chronologique soit respecté.

b) Mise en place des conteneurs de données

Il s'agit à présent de mettre en place tous les conteneurs de données de l'application. Ils seront au nombre de 4 :

- **Professeurs** : il s'agit d'une liste triée de professeurs (du type **ListeTriee<Professeur>**) triée par ordre alphabétique sur le nom et le prénom
- **Groupes** : il s'agit d'une liste triée des groupes (du type **ListeTriee<Groupe>**) triée par ordre croissant du numéro de groupe
- **Locaux** : il s'agit d'une liste triée des locaux (du type **ListeTriee<Local>**) triée par ordre décroissant du nombre de places
- **Courss** : il s'agit d'une liste triée de tous les cours planifiés (du type **ListeTriee<Cours>**) triée par ordre chronologique.

Tous ces conteneurs devront être manipulés à l'aide l'itérateur développé dans les jeux de tests. Ceci sera vérifié lors de l'évaluation finale.

c) Accès direct à l'horaire d'un planifiable : les « SmartPointer »

Dans l'état actuel des choses, tous les cours donnés par un professeur se trouvent dans le conteneur global **Courss**. Il faut donc parcourir tout ce conteneur pour retrouver les cours de ce professeur, ce qui n'est pas particulièrement efficace. Il serait judicieux qu'on puisse accéder directement aux seuls cours concernant ce professeur. La même remarque peut être faite pour un groupe ou un local.

L'idée est donc d'ajouter à la classe Planifiable une liste triée ne contenant que les Cours concernant ce planifiable. Cependant, les événements seraient ainsi stockés en quatre exemplaires, ce qui n'est pas efficace non plus. La solution consiste donc à ajouter à la classe Planifiable, non pas une liste triée de Cours, mais bien une liste triée de pointeurs vers les Cours concernés dans le conteneur Courss. Le problème à présent est que cette liste sera triée sur les valeurs de pointeurs et non chronologiquement…

La solution à ce dernier problème consiste donc à encapsuler ce pointeur dans une classe généralement connue sous le nom de « SmartPointer » et dont les opérateurs de comparaison sont surchargés de telle sorte que l'on compare les objets pointés et non les valeurs de pointeur.

On vous demande donc de développer la classe **template SmartPointer** dont la structure est la suivante :

```
template<class T> class SmartPointer
{
  private:
    T * val;
…
};
```

Et qui contient les méthodes suivantes

- Constructeurs par défaut, d'initialisation et de copie,
- Un destructeur. Mais attention ! Ce destructeur ne doit pas libérer la zone mémoire de l'objet T pointé par val, car celle-ci est peut-être statique.
- Une méthode Delete() qui libère la mémoire associée à l'objet T pointé
- Une méthode T* getValeur() qui retourne la valeur du pointeur
- La surcharge de l'opérateur * retournant l'objet T pointé par val
- Les opérateurs de comparaison <, >, ==, … permettant de comparer les objets pointés (et non les pointeurs) en faisant appel aux opérateurs de comparaison de la classe T

Vous pouvez à présent ajouter à la classe Planifiable la variable membre suivante :

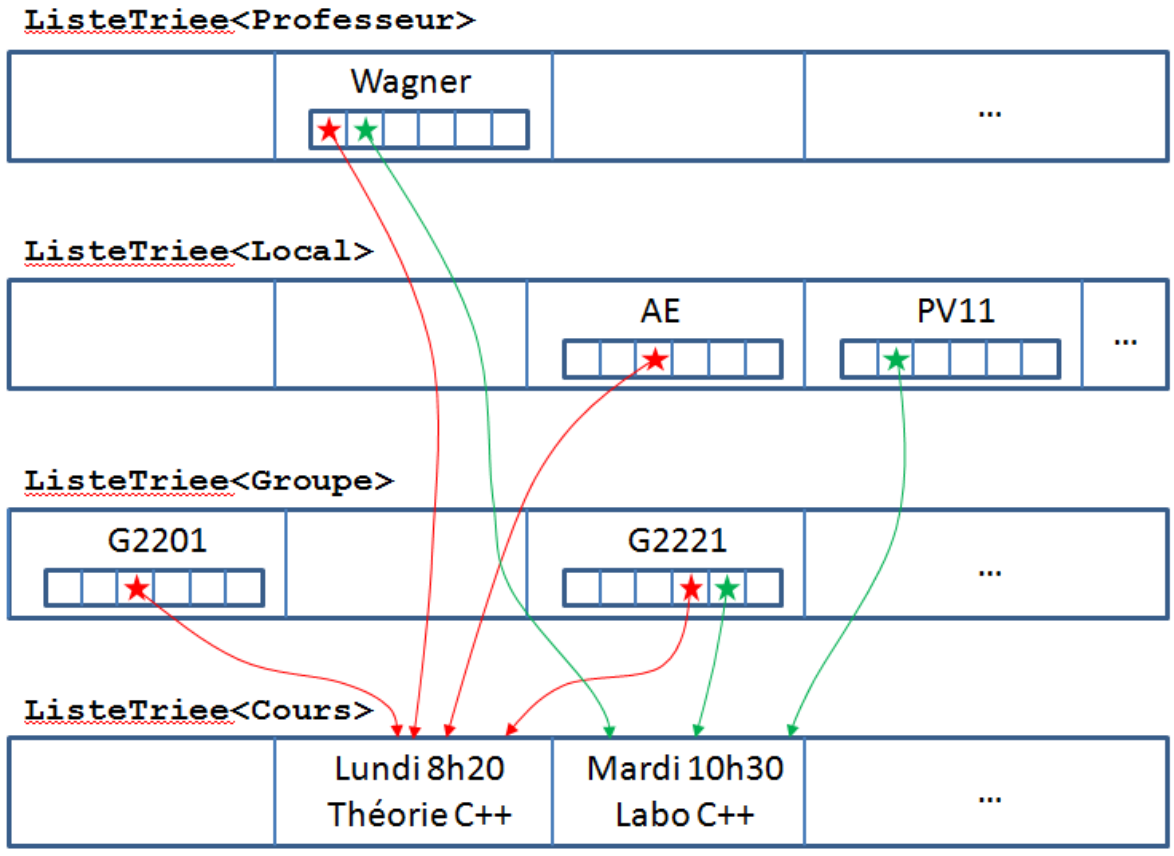
- **horaire** : variable du type **ListeTriee<SmartPointer<Cours>>** qui contient les pointeurs vers tous les cours concernant un planifiable, cours situés dans le conteneur global Courss

ainsi que les méthodes d'instances suivantes :

- **AjouteCours(const Cours *pC)** qui permet d'ajouter le cours pointé par pC dans le conteneur Horaire
- **SupprimeCours(const Cours *pC)** qui permet de supprimer le cours pointé par pC du conteneur Horaire

d) Schéma général de l'architecture de l'application

Le schéma suivant présente les différents conteneurs de données, ainsi que les pointeurs allant des planifiables vers les cours associés :



Sur ce schéma, nous voyons deux cours en exemple :

- Lundi 8h20 : Théorie C++ donné par Wagner, à l'AE, aux groupes 2201 et 2221
- Mardi 10h30 : Labo C++ donné par Wagner, au PV11, au groupe 2221

d) Sérialisation de l'horaire

Toute l'information concernant l'horaire sera enregistrée dans un fichier binaire (par exemple « 1erQuadrimestre_2015.hor ») dans lequel seront stockés séquentiellement :

- Les professeurs contenus dans la liste triée des professeurs,
- Les locaux contenus dans la liste triée des locaux
- Les groupes contenus dans la liste triée des groupes
- Les cours contenus dans la liste triée des cours

Attention, la variable **horaire** des planifiables (**ListeTriee<SmartPointer<Cours>>**) ne devra pas être enregistré sur disque. Les différents pointeurs seront re-générés au fur et à mesure de la lecture des cours dans le fichier.

e) Une classe encapsulant tout l'horaire (BONUS)

Comme on l'a fait pour le fichier binaire .hor, on pourrait imaginer de regrouper toute l'information d'un horaire au sein d'une même classe :

```
class Horaire
{
   private :
      ListeTriee<Professeur> professeurs ;
      ListeTriee<Groupe> groupes ;
      ListeTriee<Local> locaux ;
      ListeTriee<Cours> cours ;

   public :
      …
} ;
```

L'idée est alors de mettre en place toute une série de méthodes publiques permettant de gérer les données :

- void save(const char *nomFichier) ;
- void load(const char* nomFichier) ;
- void planifie(const Cours& cours) ;
- …

Le programme principal ne ferait alors qu'instancier une instance de cette classe et manipulerait tout l'horaire à l'aide des méthodes publiques de la classe.

3.4 Fonctionnalité des items du menu

Nous pouvons à présent aborder les différentes fonctionnalités du menu. Avant de commencer à planifier des cours, nous devons tout d'abord disposer de planifiables. Nous allons donc commencer par le menu les concernant.

a) Gestion des planifiables

Au démarrage de l'application, tous les conteneurs de données sont vides. Il est donc nécessaire d'ajouter des professeurs, groupes et locaux avant de pouvoir planifier le moindre cours. Voici donc le détail du menu permettant de gérer cela :

- L'item 5 permet d'ajouter un professeur ou un groupe ou un local dans le conteneur correspondant. Toutes les informations concernant ce planifiable sont demandées à l'utilisateur. Attention, on veillera à ne pas créer un nouveau professeur ayant le même identifiant qu'un professeur existant déjà. Il en va de même pour les groupes et les locaux.
- L'item 6 permet d'importer un ensemble de professeurs ou groupes ou locaux à partir d'un fichier csv (trois fichiers csv vous seront fournis au laboratoire) et de les ajouter au conteneur correspondant.
- L'item 7 permet d'afficher la liste des professeurs ou groupes ou locaux dans l'ordre imposé par leur conteneur.
- L'item 8 permet de supprimer un professeur ou un groupe ou un local de son conteneur respectif. Attention, on ne pourra supprimer un planifiable que s'il n'est concerné par aucun cours déjà planifié. Par exemple, un professeur fraichement créé ou importé n'a pas de cours planifié, il pourrait donc être supprimé.

b) Gestion de l'horaire

Nous pouvons à présent aborder les items correspondant au menu de gestion de l'horaire :

- L'item 9 permet de planifier un cours. Il faut tout d'abord demander à l'utilisateur toutes les informations nécessaires : jour, heure, durée, local, professeur, groupe(s). Une fois ces informations encodées, il est nécessaire de vérifier la présence de chaque planifiable dans les conteneurs correspondant. En effet, on ne pourra pas planifier un cours pour un planifiable non-existant. Ensuite, il faudra vérifier la disponibilité de chaque planifiable pour le timing donné (il serait judicieux d'ajouter une méthode **bool estDisponible(const Timing &t)** à la classe Planifiable). En effet, on ne pourra pas planifier deux cours en même temps pour un planifiable. Finalement, on pourra instancier l'objet Cours, l'insérer dans le conteneur Courss et mettre à jour l'horaire de chaque planifiable concerné (les pointeurs vers l'objet Cours créé).
- L'item 10 permet de déplanifier un cours dont on demande le code à l'utilisateur. Il s'agit de le supprimer de la liste triée des cours et mettre à jour l'horaire de chaque planifiable (suppression des pointeurs vers ce cours).
- L'item 11 permet de déplanifier tous les cours d'un professeur ou un groupe ou un local, et dont l'identifiant est demandé à l'utilisateur.
- L'item 12 permet d'afficher tous les cours, dans l'ordre chronologique, d'un professeur ou un groupe ou un local dont l'identifiant est demandé à l'utilisateur.

- L'item 13 permet d'afficher tous les cours, dans l'ordre chronologique, tous professeurs, groupes et locaux confondus, pour un jour de la semaine demandé à l'utilisateur.

c) Publier l'horaire

Ce sous-menu ne comporte que l'item 14 qui consiste à exporter l'horaire d'un professeur ou un groupe ou un local (dont on demande l'identifiant à l'utilisateur) au format texte. Par exemple, pour le professeur Wagner, on pourra avoir, en sortie le fichier « Wagner_Jean-Marc.txt » contenant

```
Horaire de Jean-Marc Wagner (1erQuadrimestre_2015)
LUNDI :
   8h20 (2h00,AE) Théorie C++ : G2201, G2221
   13h30 (1h30,LE0) Labo UNIX : G2223
MARDI :
   10h30 (2h00,PV11) Labo C++ : G2221
   …
MERCREDI :
   …
JEUDI :
   …
VENDREDI :
   …
```

Et pour le groupe 2221, un fichier "G2221.txt" :

```
Horaire de G2221 (1erQuadrimestre_2015)
LUNDI :
   8h20 (2h00,AE) Théorie C++ : Wagner Jean-Marc
   …
MARDI :
   10h30 (2h00,PV11) Labo C++ : Wagner Jean-Marc
   …
…
```

d) Menu Fichier

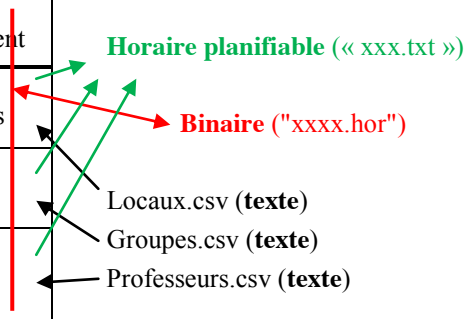
Nous pouvons à présent revenir sur le menu fichier gérant l'enregistrement et la lecture d'un horaire sur disque :

- L'item 1 permet de créer un nouvel horaire dont le nom sera demandé à l'utilisateur. Par concaténation avec l'extension « .hor », on obtiendra ainsi le nom du fichier binaire correspondant. On ne pourra pas créer un nouvel horaire tant qu'il y aura un horaire ouvert en cours. Il faudra alors tout d'abord le fermer.
- L'item 2 permet de charger un horaire en mémoire à partir d'un fichier dont le nom est demandé à l'utilisateur. Avant de pouvoir ouvrir un horaire, l'utilisateur devra au préalable fermer l'horaire en cours. Ensuite, le chargement du fichier permettra de remplir les conteneurs de données. Attention ! N'oubliez pas de remettre à jour les listes de pointeurs des planifiables.
- L'item 3 permet d'enregistrer l'horaire en cours (dont le nom est connu) sur disque (dans un seul fichier binaire dont la structure a été expliquée plus haut).
- L'item 4 permet de fermer l'horaire en cours, c'est-à-dire de vider tous les conteneurs de données (professeurs, groupes, locaux et cours)

3.5 Conteneurs de données et fichiers : résumé

Vous trouverez ci-dessous un tableau récapitulatif rappelant les types de containers utilisés pour gérer en mémoire les différents ensembles d'objets ainsi que les types de fichiers à utiliser.

Données	*Classe*	*Container*	*Fichiers*
Courss	**Cours**	**Liste triée** chronologiquement	
Locaux	**Locaux**	**Liste triée** par nombre de places	
Groupes	**Groupe**	**Liste triée** par numéro	
Professeurs	**Professeur**	**Liste triée** par identifiant	



3.6 Fichiers Professeurs.csv, Groupes.csv et Locaux.csv

Les fichiers textes Professeurs.csv, Groupes.csv et Locaux.csv sont disponibles dans le centre de ressources de Jean-Marc Wagner. Au départ, il s'agit de fichiers Excel qui ont été enregistrés au format csv. Ces fichiers peuvent s'éditer avec Excel mais également avec "Notepad" ou n'importe quel éditeur de texte. Une ligne du fichier texte correspond à une ligne dans le fichier Excel, les colonnes sont séparées par des ";".