Laboratoire de Programmation en C++

**2ème informatique et systèmes :
option(s) industrielle et réseaux (1er et 2ème quart)
et 2ème informatique de gestion (1er, 2ème et 3ème quart)**

Année académique 2014-2015

Gestion d'une concession automobile

Véronique Jacquet
Denys Mercenier
Claude Vilvens
Jean-Marc Wagner

1. Introduction

1.1 Le contexte : la gestion d'une concession automobile

Les travaux de Programmation Orientée Objets (POO) se déroulent dans le contexte de la gestion d'une concession automobile. Plus particulièrement, il s'agit de gérer l'interaction entre les vendeurs de la concession et les acheteurs potentiels, mais également les contrats de vente. En vue de simplifier les choses, aucune gestion de stock ne sera réalisée.

Dans un premier temps, il s'agira de prévoir la gestion des différents modèles de voiture existants pour une marque donnée, ainsi que les options disponibles pour ces différents modèles. Nous aborderons ensuite, la problématique de la gestion des contrats, une fois que le client aura choisi le modèle et les options éventuelles.

Deux types d'utilisateurs auront accès à l'application finale:

- Les **vendeurs** qui réalisent, en compagnie d'un **client**, la conception d'un véhicule à partir du catalogue de la marque (**modèles** de base, **options** disponibles, prix) ; ils créent également les **contrats** de vente, et assurent leur suivi.
- Les **administratifs**, pouvant gérer les utilisateurs de l'application, mais qui peuvent également accéder aux différents contrats des vendeurs à des fins de contrôle.

1.2 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 (ainsi qu'au 3ème quart pour les étudiants d'informatique de gestion - l'énoncé correspondant sera fourni ultérieurement). 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.

1.3 Règles d'évaluation

1) L'évaluation établissant la note du cours de programmation orientée objet est réalisée de la manière suivante :

♦ théorie : un examen écrit en janvier 2015 (sur base d'une liste de questions fournies en novembre et à préparer) et coté sur 20;
♦ laboratoire (Informatique et systèmes) : 2 évaluations (aux dates précisées dans l'énoncé de laboratoire), chacune cotée sur 20; la moyenne pondérée (30% pour la première partie et 70% pour la seconde partie) de ces 2 cotes fournit une note de laboratoire sur 20;
♦ laboratoire (Informatique de gestion) : 3 évaluations (aux dates précisées dans l'énoncé de laboratoire), chacune cotée sur 20; la moyenne pondérée de ces 3 cotes (poids respectifs de 2/10, 4/10 et 4/10) fournit une note de laboratoire sur 20;
♦ note finale (toutes sections) : **moyenne 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, ainsi que lors d'une éventuelle prolongation de session.

2) 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)

3) 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**.

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

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 2015) :

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 (**Modele)**	Constructeurs, destructeurs
	Getters, Setters et méthode Affiche
	Fichiers Modele.cxx, Modele.h et makefile
• **Jeu de tests 2 :** Agrégation entre classes (classe **Voiture**)	Agrégation par valeur (classe **Modele)**
	Agrégation par référence (classe **Option**)
	Méthodes AjouteOption(), RetireOption(), getPrix()
• **Jeu de tests 3 :** Surcharge des opérateurs	Operateurs = + - de la classe Voiture
	Opérateurs < > == de la classe Voiture
	Opérateurs << et >> de Voiture, Modele et Option
	Opérateurs de post et pré-décrémentation de Option
• **Jeu de tests 4 :** Héritage et virtualité	Hiérarchie : classes **Personne, Client et Employe**
	Test des méthodes (non) virtuelles
	Variables membres statiques
• **Jeu de tests 5 :** Exceptions	**InvalidFonctionException**
	InvalidPasswordException
	Utilisation correcte de **try**, **catch** et **throw**
EVALUATION 2 (Jeux de tests 6 à 7) → Examen de Janvier 2015	
• **Jeu de test 6 :** Containers génériques	Classe **abstraite ListeBase**
	Classe **Liste** (→ int et Option)
	Classe **Pile**
	Classe **ListeTriee** (→ int et Personne)
	Itérateur : classe **Iterateur**
• **Jeu de test 7 :** Flux	Méthodes **Save()** et **Load()** de Option et Modele
	Méthodes **Save()** et **Load()** de Voiture

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 de modèle de voiture. Par exemple, Peugeot propose le modèle « 208 Access 3 Portes» qui présente un certain nombre de caractéristiques de base (cylindrée, puissance, nombre de chevaux, émission de CO2, …). Afin de simplifier les choses, nous nous limiterons aux caractéristiques essentielles.

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

- Un **nom** : une chaine de caractères allouée dynamiquement (**char ***) en fonction du texte qui lui est associé (Exemple : « 208 Access 3 Portes »).
- Une **puissance** : un entier (**int**) représentant la puissance en chevaux du moteur.
- Un booléen **diesel** (**bool**) valant « true » s'il s'agit d'un moteur diesel, ou « false » s'il s'agit d'un moteur essence.
- Le **prix de base** du modèle (**float**). Ce prix ne tient donc pas compte d'éventuelles options supplémentaires.

Comme vous l'impose le premier jeu de test (Test1.cxx), on souhaite disposer 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[], voir la suite). **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 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 Modele (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 : Associations entre classes : agrégations

a) Une agrégation par valeur (Jeu de tests : Test2a.cxx)

Il s'agit à présent de créer une classe qui permettra au vendeur de réaliser un projet de voiture pour un client. Ce projet de voiture comporte un modèle de base (avec un prix de base) et un ensemble d'options que le vendeur pourra ajouter pour rencontrer les souhaits de l'acheteur (peinture métallisée, air conditionné, vitres teintées arrière, …). Ce projet sera modélisé par la classe **Voiture** dont voici les variables membres (d'autres suivront pour les options mais une chose à la fois ☺ !) :

- Le **nom** du projet : il s'agit d'une chaine de caractères (**char ***) contenant un nom du genre « Projet_208GTI_MrDugenou ».
- Un **modèle** (variable du type **Modele**) : il s'agit du modèle de base du projet.

A nouveau (comme cela est imposé dans **Test2a.cxx**), vous devez programmer les trois types de constructeurs attendus, le destructeur et les méthodes getXXX/setXXX pour les deux variables membres. Bien sûr, cette classe doit posséder ses propres fichiers .cxx et .h.

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

b) Agrégation par référence (Jeu de tests : Test2b.cxx et Test2c.cxx)

Il s'agit à présent de pouvoir ajouter au projet de voiture un certain nombre d'options souhaitées par l'acheteur. Nous allons donc modéliser une option par la classe **Option** qui présente les variables membres suivantes :

- Un **code** : il s'agit d'une chaine de 4 caractères de taille fixe (**char [5]**) qui identifie de manière unique une option dans le catalogue (Pour simplifier les choses, nous considérerons que n'importe quelle option peut être ajoutée à n'importe quel modèle de voiture). Exemple : « AB0B », « J2U6 », …
- Un **intitulé** : il s'agit d'une chaine de caractères (**char ***) représentant l'intitulé de l'option. Exemple : « Peinture métallisée », « Air conditionné », « Jantes en alliage léger 15 pouces », …
- Un **prix** : il s'agit d'un nombre (**float**) représentant le prix de l'option.

A nouveau, vous devez programmer les constructeurs, le destructeur et les getters/setters adéquats. Dans la suite de l'énoncé, nous ne le dirons plus mais il doit être clair que **toute classe doit disposer de constructeurs, d'un destructeur (!!!) et des getters/setters adéquats**.

Nous pouvons à présent compléter notre classe **Voiture** par la variable membre suivante :

- la variable **options** : il s'agit de l'ensemble des options que l'on va ajouter au projet. Nous limiterons le nombre d'options à 10. Cette variable sera du type « **tableau de pointeurs d'options** », c'est-à-dire **Option* options[10]**. Chacune des cases de ce tableau sera initialisée à NULL mais pourra dans la suite pointer vers un objet du type Option.

Remarquons cependant que la variable options n'a pas besoin de getter/setter. Il s'agit d'un conteneur d'options dans lequel on ajoutera/retirera des options. On préférera donc ajouter à la classe **Voiture** les méthodes suivantes :

- **AjouteOption(const Option & option)** qui permet ajouter une option au projet. Pour cela, vous recherchez une case vide (NULL) dans le tableau options et vous lui assignez un pointeur vers un objet Option que vous aurez alloué dynamiquement.
- **RetireOption(const char* code)** qui permet de retirer, de la liste des options du projet, l'option dont le code est passé en paramètre à la méthode. Remarquez qu'une fois trouvé, vous devez libérer la mémoire et remettre la case du tableau à NULL…
- **getPrix()** qui permet de retourner le prix de la voiture dans l'état actuel. Il s'agit du prix de base du modèle augmenté du prix de chaque option. Remarquez que la classe Voiture ne contient pas de variable membre prix. La méthode getPrix() calcule le prix à partir du prix du modèle et du prix de chaque option présente dans le tableau options, avant de retourner le résultat de son calcul. De l'extérieur, le « programmeur utilisateur de la classe » ne le voit pas. Vive l'encapsulation ☺ !

Notez que la méthode Affiche de la classe Voiture doit être mise à jour pour tenir compte des options.

La classe Voiture possède un certain nombre de pointeurs vers des objets de type Option. Elle ne contient donc pas d'objets Option en son sein mais seulement des pointeurs vers de tels objets. On parle d'agrégation par référence.

2.3 Jeu de tests 3 : 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 conception d'un projet de voiture.

a) Surcharge des opérateurs =, + et – de la classe Voiture (Test3a.cxx)

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

```
Modele m("208 Access 3P 1.4 HDi",1398,true,14600.0f);
Voiture v1,v2("208_MrDugenou",m),v3;

v1 = v2 ;

Option op1("0MM0","Peinture metallisee",450.0f) ;
Option op2("ZH75","Jantes alliage 15 pouces”,450.0f) ;

v3 = v2 + op1 ;  // attention, v2 doit rester inchangé
v3 = v3 + op2 ;  // on ajoute l'option op2 à la voiture v3
v3.Affiche() ;
v3 = v3 – op1 ;  // on retire l'option op1 de la voiture v3
v3.Affiche() ;
```

Bien sûr, on portera une attention particulière sur le tableau de pointeurs afin que deux objets de type Voiture ne pointent pas vers les mêmes objets Option…

b) Surcharge des opérateurs de comparaison de la classe Voiture (Test3b.cxx)

Il peut être intéressant pour le client de comparer deux projets de voiture afin qu'il puisse choisir celui qui lui convient le mieux. On vous demande donc de surcharger les opérateurs <, > et == de la classe Voiture, permettant d'exécuter un code du genre

```
Voiture v1,v2 ;
…
if (v1 > v2) cout << message ;
if (v1 == v2) cout << autre message;
if (v1 < v2) …
```

c) Surcharge des opérateurs d'insertion (Test3c.cxx)

On vous demande à présent de surcharger les operateurs << et >> des classes Modele, Option et Voiture, ce qui permettra d'exécuter un code du genre :

```
Modele m;
Voiture v1;
Option op1("0MM0","Peinture metallisee",450.0f) ;
Option op2("ZH75","Jantes alliage 15 pouces",450.0f) ;

cin >> m ;
cout << m ;
cin >> v1 ;  // permet d'encoder le modèle, la référence mais pas les options
v1 = v1 + op1 ;
v1 = v1 + op2 ;
cout << "Projet de Mr Dugenou :" << v1 << endl;
cout << op1 << op2 << endl ;
```

d) Surcharge des opérateurs de pré et post-décrémentation (Test3d.cxx)

Enfin, il est possible qu'au cours de la conception d'un projet, le vendeur accepte de baisser le prix d'une option pour faire plaisir au client et s'assurer de la réussite de la vente ☺. On vous demande donc de programmer les operateurs post et pré-décrémentation de la classe Option. Ce qui permettra d'exécuter le code suivant :

```
Option op1("0MM0","Peinture metallisee",450.0f) ;
Option op2("ZH75","Jantes alliage 15 pouces",450.0f) ;

cout << --op1 << endl ;
cout << op2-- << endl ;
cout << op2 << endl ;
```

Une décrémentation réduira le prix d'une option de 50,00 euros.

2.4 Jeu de tests 4 (Test4.cxx) :

Associations de classes : héritage et virtualité

Nous allons à présent aborder la modélisation des personnes intervenant dans notre application. Trois types de personne interviennent :

- Les clients qui ont un nom, un prénom, une adresse et un numéro de client.
- Les vendeurs qui sont des employés de la concession automobile et qui ont un nom, un prénom, un numéro d'employé, et un couple (login-mot de passe) pour pouvoir utiliser l'application.
- Les administratifs qui sont des employés de la concession automobile et qui ont un nom, un prénom, un numéro d'employé, et un couple (login-mot de passe) pour pouvoir utiliser l'application.

a) Héritage et hiérarchie

On remarque tout de suite que les trois intervenants ont un point commun, ce sont des personnes (on s'en serait douté ☺). L'idée est alors de concevoir une hiérarchie de classes, par héritage, dont la classe de base regroupera les caractéristiques communes aux trois intervenants, à savoir leur nom et prénom. Cette classe de base sera la classe **Personne** et contiendra (dans un souci de simplification) les deux variables membres suivantes :

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

client est une personne mais qui a, en plus, un numéro de client et une adresse. On vous demande donc de programmer la classe **Client**, qui hérite de la classe Personne, et qui présente, en plus, les variables membres suivantes :

- Un **numéro** de client: un entier (**int**) qui identifie le client de manière unique.
- Une **adresse** : chaine de caractères (**char ***).

Vendeur et administratifs ne sont pas tellement différents par leurs caractéristiques, ce qui les différencie, c'est leur fonction. Donc, on vous demande de programmer la classe **Employe**, qui hérite de la classe Personne, et qui présente, en plus, les variables membres suivantes :

- Un **numéro** d'employé : un entier (**int**) qui identifie l'employé de manière unique.
- Un **login** : chaine de caractères (**char ***).
- Un **mot de passe** : chaine de caractères (**char ***).
- Une **fonction** : chaine de caractères (**char ***) pouvant contenir « Vendeur » ou « Administratif » selon le cas.

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

b) Virtualité

Afin de vous faire comprendre la notion de virtualité et donc d'être capable différencier les méthodes virtuelles et les méthodes non virtuelles, on vous demande d'ajouter aux trois classes de la hiérarchie (Personne, Client et Employe) les deux méthodes suivantes :

- **void Affiche()** : qui affiche les caractéristiques correspondantes mais **qui doit être non-virtuelle, et ce, pour des raisons purement pédagogiques**.
- **char* toString()** : qui retourne une chaine de caractères décrites plus bas et **qui doit être virtuelle**.

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

Vendeur	→	« VENDEUR:numero#nom »
Client	→	« CLIENT:numero#nom »
Administratif	→	« ADMINISTRATIF:numero#nom »
Personne	→	« Nom#Prenom »

c) Variables membres statiques

La variable membre **fonction** de la classe Employe ne peut pas prendre n'importe quelle valeur ; elle ne peut prendre que les valeurs « Vendeur » ou « Administratif ». Donc, on vous demande d'ajouter à la classe Employe **deux variables membres statiques** du type **chaines de caractères (char[])**. Celles-ci s'appelleront ADMINSTRATIF et VENDEUR et auront pour valeur respective « Administratif » et « Vendeur ».

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 :

- **InvalidFonctionException** : lancée lorsque l'on tente d'affecter à un employé une fonction qui ne correspond à aucune des deux variables statiques ADMINISTRATIF ou VENDEUR. Celle-ci peut apparaître lors de la construction de l'objet ou lors de l'initialisation de celui-ci à l'aide d'un setter. Par exemple, si e est un employé, e.setFonction(« BigBoss ») lancera une exception car « BigBoss » n'est pas une fonction reconnue par l'application. La classe **InvalidFonctionException** contiendra une seule variable membre du type **chaine de caractères** (**char ***) et qui contiendra un message lié à l'erreur, comme par exemple « Fonction invalide ! ».

- **InvalidPasswordException** : lancée lorsque l'on tente d'affecter à un employé un mot de passe dont la longueur est inférieure à 8 et/ou ne contient pas au moins une lettre et un chiffre. La classe **InvalidPasswordException** contiendra une seule variable membre de type **int** dont la valeur sera égale à la longueur du mot de passe ayant provoqué l'erreur. Plus précisément,
  - Si cette valeur est strictement inférieure à 8, cela signifie que le mot de passe est trop court.
  - Si cette valeur est supérieure ou égale à 8, cela signifie que le mot de passe a une longueur correcte mais qu'il ne contient pas au moins une lettre et un chiffre.

  En cas d'erreur, le test de cette valeur permettra de connaître la cause exacte de l'erreur. Exemples : Si e est un employé, l'appel de e.setMotDePasse(« 12345 ») lancera une InvalidPasswordException dont la valeur est 5, signifiant que le mot de passe est trop court. Par contre, l'appel de e.setMotDePasse(« azertyuio ») lancera une InvalidPasswordException dont la valeur est 9, signifiant que le mot de passe a une longueur suffisante (9) mais qu'il ne contient pas au moins une lettre et un chiffre.

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 :

Les containers et les templates

a) L'utilisation future des containers

On conçoit sans peine qu'une application de gestion d'une concession automobile va utiliser des containers mémoire divers qui permettront par exemple de contenir tous les clients ou tous les employés de la concession. 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 **vide()** 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** **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, …). Le fait que cette méthode soit virtuelle pure rend la classe ListeBase **abstraite**, et vous obligera à la redéfinir dans toute classe dérivée vouée à être instanciée.

c) Une première classe dérivée : La liste simple (Test6a.cxx)

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 **Option**.

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

d) Une seconde classe dérivée : La Pile (Test6b.cxx)

Afin d'illustrer « l'efficacité » de la hiérarchie ainsi mise en place, on vous demande de créer la classe **Pile** qui hérite de la classe ListeBase et qui

- redéfinit la méthode **insere** de telle sorte que l'élément inséré se place en tête de liste (il s'agit du sommet de la pile).
- définit la méthode **retire()** qui retire un élément du sommet de la pile, c'est-à-dire celui situé en tête de liste.

e) La liste triée (Test6c.cxx)

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 **Personne**. Celles-ci devront être triées par ordre alphabétique sur le nom et le prénom.

f) Parcourir et modifier une liste : l'itérateur de liste (Test6d.cxx et Test6e.cxx)

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 la référence de l'élément 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.

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**).

a) <u>La classe Voiture se sérialise elle-même</u>

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

- **Save()** permettant d'enregistrer dans un fichier toutes les données de la voiture (nom du projet, modèle et options) et cela champ par champ. Ce fichier sera un fichier **<u>binaire</u>** (utilisation des méthodes **write** et **read**) dont le nom sera obtenu par la concaténation du nom du projet avec l'extension « .car ». Exemple : « Projet208_MrDugenou.car ».

- **Load(const char* nomFichier)** permettant de charger toutes les données relatives à une voiture enregistrée dans le fichier dont le nom est 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 **Modele** et **Option** 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 Voiture lorsqu'elle devra enregistrer ou lire sa variable membre de type Modele et/ou les différentes options.

3. Deuxième Partie : Développement de l'application

Points principaux de l'évaluation	subdivisions
EVALUATION 2 (Développement de l'application) → Examen de Janvier 2015	
• **Menu Administratif :** Gestion des utilisateurs	Fonctionnalités respectées ?
	Instanciation et utilisation de **ListeTriee<Employe>**
	Gestion du **fichier binaire** « Utilisateurs.dat » → Entrée en session
• **Menu Vendeur :** Gestion des clients	Fonctionnalités respectées ?
	Instanciation et utilisation de **ListeTriee<Client>**
	Gestion du **fichier binaire** « Clients.dat »
• **Menu Vendeur :** Gestion des projets de voiture	Fonctionnalités respectées ?
	Instanciation et utilisation de **ListeTriee<Modele>** et de **Liste<Option>**
	Lecture et parsing des **fichiers textes** (.csv)
	Gestion des **fichiers binaires** *.car (**un par projet**)
• **Menu Vendeur :** Gestion des contrats	Fonctionnalités respectées ?
	Classe **Contrat**
	Instanciation et utilisation de **ListeTriee<Contrat>**
	Gestion du **fichier binaire** « Contrats.dat »
• **Menu Administratif :** Gestion des contrats	Fonctionnalités respectées ?
	Calcul du chiffre d'affaire d'un vendeur
• **Utilisation des briques de base**	Utilisation de l'itérateur **ListeBaseIter**
	Utilisation de **InvalidFonctionException** et de **InvalidPasswordException**
	Création et utilisation d'autres exceptions ?

3.1 Les utilisateurs de l'application : introduction

Comme déjà dit, l'application vise 2 profils d'utilisateurs. Nous trouverons donc :

- Les administratifs qui gère les utilisateurs de l'application mais qui ont également une vue sur les contrats conclus par les vendeurs.
- Les vendeurs qui gèrent les clients, la conception des projets de voiture mais également les contrats passés avec les clients.

Les données relatives aux utilisateurs (classe **Employe**) sont enregistrées dans un fichier **binaire** qui s'appellera « Utilisateurs.dat » et qui contiendra les données de tous les utilisateurs. Celui-ci est géré de façon très simplifiée. Il est chargé en mémoire au lancement de l'application dans une liste triée (classe **ListeTriee**) d'employés triés sur le numéro d'employé. Cette liste est enregistrée dans le fichier lorsque l'application se termine.

Au démarrage, l'application demande à l'utilisateur de s'identifier au moyen de son login et mot de passe. Un exemple est présenté ci-dessous.

```
**********************************
********** Bienvenue !!! **********
**********************************

Login : Merce
Mot de passe : azerty34
```

Après validation, le développeur veillera à parcourir sa liste (utilisation de l'itérateur **ListeBaseIter**) d'utilisateurs et utilisera le champ « fonction » pour adapter le menu en conséquence. En effet, un vendeur ne dispose pas du même menu qu'un administratif.

3.2 Le menu des administratifs

Le menu sera du type suivant :

```
**************************************************
********** Menu Administratif ********************
**************************************************
      0. Changer de mot de passe
Gestion des utilisateurs ****************************
      1. Afficher la liste des utilisateurs
      2. Afficher les infos d'un utilisateur
      3. Créer un nouvel utilisateur
      4. Réinitialiser le mot de passe d'un utilisateur
Gestion des contrats ********************************
      5. Afficher tous les contrats
      6. Afficher les détails d'un contrat
      7. Afficher les contrats et le chiffre d'affaire d'un vendeur

      Q. Quitter l'application
```

Tout d'abord, l'item 0 du menu permet à un utilisateur de changer son propre mot de passe. Nous allons tout d'abord décrire les items du menu de gestion des utilisateurs. La gestion des contrats de vente par les administratifs sera abordée plus loin.

a) Gestion des utilisateurs

Ce premier menu permet de gérer la liste des utilisateurs pouvant accéder à l'application. Au démarrage de l'application, si le fichier Utilisateurs.dat n'existe pas, celui-ci est bidonné avec un utilisateur du type Administratif ayant le login « admin » et le mot de passe « admin ».

Passons à présent en revue les différents items du menu :

- L'item 1 permet d'afficher la liste de tous les utilisateurs de l'application (on n'affichera que les informations essentielles de chaque utilisateur.
- L'item 2 permet d'afficher les informations détaillées d'un utilisateur dont on fournit le nom.
- L'item 3 permet d'encoder les caractéristiques d'un nouvel utilisateur (à l'exception du mot de passe) et de l'ajouter à la liste triée des utilisateurs.
- L'item 4 permet de réinitialiser le mot de passe d'un utilisateur dont on fournit le login. Pour cela, il suffit de remettre le pointeur motDePasse de l'objet Employe à NULL.

On remarquera que lorsque l'on crée un nouvel utilisateur, on ne précise pas son mot de passe. C'est lors de sa première entrée en session que l'application lui demandera d'encoder un mot de passe et de le confirmer ensuite. Bien sûr, lors de cet encodage, la gestion de l'exception **InvalidPasswordException** doit être gérée. De même, lorsque l'on crée un nouvel utilisateur, la gestion de l'exception **InvalidFonctionException** doit être faite.

3.3 Le menu des vendeurs

Le menu sera du type suivant :

```
*****************************************
********** Menu Vendeur ********************
*****************************************
      0. Changer de mot de passe
Gérer les clients *************************
      1. Ajouter un nouveau client
      2. Supprimer un client
      3. Afficher la liste des clients
Gérer les projets de voiture ***************
      4. Afficher la liste des modèles
      5. Afficher la liste des options
      6. Nouvelle Voiture
      7. Charger une voiture
      8. Afficher la voiture en cours
      9. Ajouter une option à la voiture en cours
      10. Retirer une option à la voiture en cours
      11. Appliquer une ristourne à une option de la voiture en cours
      12. Enregistrer la voiture en cours
Gérer les contrats de vente ****************
      13. Nouveau contrat
      14. Afficher tous mes contrats
      15. Afficher un contrat et le prix de vente définitif
      16. Modifier un contrat

      Q. Quitter l'application
```

Tout d'abord, l'item 0 du menu permet à un utilisateur de changer son propre mot de passe. Passons à présent en revue les autres items du menu.

a) Gestion des clients

Les clients de la concession seront stockés en mémoire dans une liste triée de clients (**ListeTriee<Client>**), triés sur leur nom et prénom. Tous les clients seront enregistrés dans un seul et même fichier binaire appelé « clients.dat ». Ce fichier sera chargé en mémoire au lancement de l'application et enregistré sur disque à la fin de l'application.

Nous pouvons à présent expliciter les items suivants :

- L'item 1 permet d'encoder un nouveau client et de l'insérer dans la liste triée des clients.
- L'item 2 permet de supprimer un client, dont on fournit le numéro, de la liste des clients. Remarquez qu'un client ayant conclu un contrat ne pourra être supprimé de la liste (voir plus loin).

- L'item 3 permet d'afficher la liste de tous les clients.

b) Gestion des projets de voiture

Pour concevoir un projet de véhicule automobile en compagnie du client, le vendeur doit disposer du catalogue de la marque. Celui-ci est fourni sous la forme de deux fichiers textes au format .csv (« Modeles.csv » et « Options.csv » ; ces fichiers vous seront fournis par votre professeur de laboratoire). Au démarrage de l'application, la lecture de ces deux fichiers textes permettra d'alimenter deux conteneurs mémoire, à savoir une liste triée de modèles, triés par ordre croissant des prix de base (**ListeTriee<Modele>**), et une simple liste d'options (**Liste<Option>**). En mémoire, on ne pourra gérer qu'un seul objet Voiture à la fois, il s'agira de la voiture en cours de conception.

Nous pouvons donc à présent décrire les items de menu suivants :

- L'item 4 permet d'afficher la liste des modèles disponibles (affichage du contenu de la liste triée de modèles).
- L'item 5 permet d'afficher la liste des options disponibles.
- L'item 6 permet de créer un nouveau projet en cours (objet Voiture) en fournissant un nom au projet et un modèle de base. Ce modèle sera obtenu dans la liste triée citée plus haut.
- L'item 9 permet d'ajouter au projet en cours une option dont on fournit le code.
- L'item 10 permet de retirer du projet en cours l'option dont on fournit le code.
- L'item 11 permet d'appliquer une ristourne à l'option (→ utilisation de l'opérateur --) du projet en cours dont on fournit le code.
- L'item 12 permet d'enregistrer le projet de voiture en cours dans un fichier binaire dont le nom est obtenu par la concaténation du nom du projet et de l'extension « .car » (→ utilisation de la méthode Save() de Voiture).
- L'item 7 permet de charger un projet en cours à partir d'un fichier binaire (d'extension .car) dont on fournit le nom.
- L'item 8 permet d'afficher un projet en cours, avec les options et le prix « options incluses ».

c) Gestion des contrats de vente (par les vendeurs)

Un contrat de vente est un document établi par un vendeur lors de l'achat d'un véhicule par un client. Une fois que le projet de véhicule a été élaboré par le vendeur, il peut ensuite passer à la création du contrat de vente. Un contrat de vente sera donc représenté par la classe **Contrat** qui présente les variables membres suivantes :

- Un **numéro de contrat**, variable de type **int**, identifiant le contrat de manière unique.
- Un **numéro de vendeur**, variable de type **int**, identifiant le vendeur qui a conclu le contrat.
- Un **numéro de client**, variable de type **int**, identifiant le client.

- Une **date**, de type **Date**, précisant la date de création du contrat. Le type **Date** est une classe que l'on vous demande de programmer. Celle-ci comportera trois variables membres de type **int** représentant le jour, le mois et l'année.
- Un **pointeur de type Voiture*** pointant vers un objet (alloué dynamiquement) de type **Voiture** représentant le projet de voiture que le client souhaite acheter.
- Une **ristourne**, variable du type **float**, représentant une éventuelle ristourne supplémentaire accordée par le vendeur.

Les contrats seront stockés en mémoire dans une liste triée par ordre chronologique (**ListeTriee<Contrats>**), enregistrés sur disque dans un seul et même fichier binaire appelé « Contrats.dat » et chargés en mémoire au lancement de l'application. Remarquez que lorsqu'un contrat est enregistré sur disque, on ne doit pas enregistrer l'objet voiture référencé par le pointeur. Il suffit d'enregistrer le nom du fichier .car correspondant (nom du projet). Lors de la relecture du contrat, le nom du fichier permettra de l'ouvrir et de récupérer l'objet de type Voiture.

Nous pouvons à présent décrire les derniers items de menu suivants :

- L'item 13 permet d'encoder les caractéristiques d'un contrat et de l'insérer dans la liste triée. Lors de l'encodage, on fournira le nom du fichier du projet (fichier d'extension .car). Celui-ci permettra de charger en mémoire le projet de voiture choisi par le client.
- L'item 14 permet d'afficher tous les contrats (uniquement les caractéristiques principales) du vendeur actuellement en session dans l'application.
- L'item 15 permet d'afficher les détails d'un contrat (ainsi que le prix final de la vente ; celui-ci correspond au prix du prix projet options incluses moins la ristourne supplémentaire accordée par le vendeur) dont on fournit le numéro.
- L'item 16 permet de modifier certaines caractéristiques d'un contrat dont on fournit le numéro. En particulier, le vendeur ne pourra modifier que la ristourne et le projet de voiture en fournissant un nouveau fichier .car.

d) Gestion des contrats de vente (par les administratifs)

Maintenant que les contrats ont été décrits en détails, nous pouvons terminer de décrire les items de menu de gestion des contrats par les administratifs :

- L'item 5 permet d'afficher tous les contrats conclus par tous les vendeurs (uniquement les caractéristiques principales).
- L'item 6 permet d'afficher les détails d'un contrat dont on fournit le numéro.
- L'item 7 permet d'afficher tous les contrats (uniquement les caractéristiques principales) d'un vendeur dont on fournit le nom mais également son chiffre d'affaire, c'est-à-dire la somme de toutes ses ventes.

3.4 Containers et persistance : résumé

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

Tous les fichiers sont chargés en mémoire, dans les containers, au démarrage, à l'exception des fichiers .car chargé à la demande par l'utilisateur. Ils sont réenregistrés à partir des containers mis à jour lorsque l'utilisateur quitte l'application.

Données	***Classe***	***Container***	***Fichier***
Utilisateurs	**Employe**	**Liste triée** par numéro d'**Employe**	**Binaire** ("utilisateurs.dat")
Modèles	**Modele**	**Liste triée** par prix de base de **Modele**	**Texte** ("Modeles.csv ")
Options	**Option**	**Liste** d'**Option**	**Texte** ("Options.csv ")
Projet de voiture en cours	**Voiture**	Une variable de type **Voiture**	Un fichier **binaire** (.car) par projet
Clients	**Client**	**Liste triée** par nom/prénom de **Client**	**Binaire** ("clients.dat")
Contrats	**Contrat**	**Liste triée** par date de **Contrat**	**Binaire** ("contrats.dat")

3.5 Fichiers Modeles.csv et Options.csv

Les fichiers textes Modeles.csv et Options.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 ";".