

Laboratoire de Programmation en C++

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

Année académique 2010-2011

RacingWorldChampionship



Véronique Jacquet
Mounawar Madani
Cécile Moitroux
Denys Mercenier
Alfonso Romio
Claude Vilvens
Jean-Marc Wagner

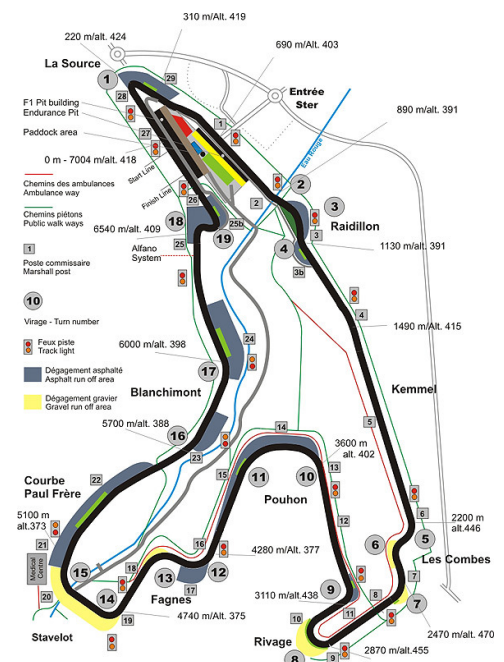


Table des matières

1 INTRODUCTION.....	3
1.1 LE CONTEXTE : UNE COMPETITION DE SPORT MOTEUR	3
1.2 PHILOSOPHIE DU LABORATOIRE	3
1.3 CONTRAT PEDAGOGIQUE.....	4
2 ENONCÉ DU 1^{ER} QUART : CRÉATION DE DIVERSES BRIQUES DE BASE NÉCESSAIRES À L'ÉLABORATION DE RACINGWORLDCHAMPIONSHIP	5
2.1 UNE PREMIERE CLASSE.....	6
2.1.1 <i>Description des fonctionnalités de la classe</i>	6
2.1.2 <i>Méthodologie de développement</i>	6
2.2 ASSOCIATIONS DE CLASSES : AGRÉGATION.....	7
2.2.1 <i>Une agrégation par valeur (ou composition)</i>	7
2.2.2 <i>Une agrégation par référence</i>	7
2.2.3 <i>Les variables membres statiques</i>	7
2.3 EXTENSION DES CLASSES EXISTANTES : SURCHARGE DES OPERATEURS	8
2.4 ASSOCIATIONS DE CLASSES : HÉRITAGE ET AGRÉGATION	9
2.4.1 <i>Héritage</i>	9
2.4.2 <i>Hiérarchie</i>	9
2.5 LES EXCEPTIONS	10
2.6 LES CONTAINERS.....	10
2.6.1 <i>L'utilisation future des containers</i>	10
2.6.2 <i>Le container typique : le vecteur</i>	10
2.7 PREMIÈRE UTILISATION DES FLUX	11
2.7.1 <i>Un fichier traceur de mouvements</i>	11
2.7.2 <i>Un petit fichier à enregistrements</i>	11
3 ENONCÉ DU 2^{ÈME} QUART : DÉVELOPPEMENT DE L'APPLICATION RACINGWORLDCHAMPIONSHIP	12

1 Introduction

1.1 Le contexte : une compétition de sport moteur

Les travaux de Programmation Orientée Objets (POO) sont sensés se dérouler dans le contexte de la gestion d'une compétition de sport moteur (voiture ou moto), "RacingWorldChampionship". Divers aspects de la gestion de celle-ci seront envisagés.

Dans un premier temps, il s'agira de prévoir la gestion des courses et des circuits sur lesquels elles sont effectuées. Ensuite, nous aborderons la gestion des différentes personnes impliquées dans de telles compétitions pour terminer par l'application finale qui permettra, à terme, d'encoder les résultats des courses, de produire le classement qui permettra d'attribuer un prix au champion de la compétition.

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

- l'administrateur de l'application
- les administrateurs des fédérations de sport moteur qui gèrent les compétitions
- les administrateurs des équipes qui s'inscrivent aux compétitions

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 1^{er} quart puis de conforter vos acquis au 2^{ème} quart (au 3^{ème} quart aussi 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 élaborer d'abord "sur papier" (donc sans programmer directement) les divers scénarios correspondant aux fonctionnalités demandées.

1.3 Contrat pédagogique

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

- ♦ théorie : un examen écrit en janvier 2011 (sur base d'une liste de questions fournies en novembre et à préparer) et coté sur 20;
- ♦ laboratoire : 2 évaluations (aux dates précisées ci-dessous) dont la cote finale est sur 20;
- ♦ note finale : **moyenne de la note de théorie (50%) et de la note de laboratoire (50%).**

Cette procédure est d'application tant en 1^{ère} qu'en 2^{ème} session, ainsi que lors d'une éventuelle prolongation de session.

2) Chacun des membres d'une équipe 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 2^{ème} 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

2	Enoncé du 1^{er} quart : Création de diverses briques de base nécessaires à l'élaboration de <i>RacingWorldChampionship</i>
----------	--

<p>Date de rentrée du dossier (papier) : le lundi 8 novembre 2010 à 8h20</p> <p>Evaluation définitive de ce dossier : à partir du 8 novembre 2010 selon les modalités indiquées par le professeur de laboratoire</p>
--

point principaux de l'évaluation	subdivisions
1. Implémentation d'une classe de base (Race)	Respect des clauses d'accès
	Séparation en fichiers .h et .cpp + main de test + instanciations dynamiques ?
	Indépendance par rapport à l'implémentation (date, circuit)
2. Agrégation entre classes	Agrégation par valeur (CDate)
	Agrégation par référence (Circuit)
	SPA_FRANCORCHAMPS, SILVERSTONE prédéfini ?
3. Surcharge des opérateurs	Gestion des longueurs de circuit, comparaisons
	Opérateurs de pré et post-incrémentation
4. Héritage et agrégation	Héritage simple : classes Person et Competitor
	Hiérarchie : Person , AuthorizedPerson (classe abstraite), FederationAdmin , TeamMember
	Test des méthodes virtuelles
5. Exceptions	InvalidDateException
	InvalidRaceSizeException
	classe de base ExceptionBase
6. Containers génériques	MyVector template en int
	MyVector template en Race et Competitor - modules séparés
7. Flux	classe FichierLog
	classe FichierCircuit

2.1 Une première classe

2.1.1 Description des fonctionnalités de la classe

Un des éléments de base de l'application est la notion de course. Il s'agit ici de créer une classe **Race** la représentant. La course est dans ce cas étudiée dans le sens le plus large du terme, donc non dédiée à une compétition en particulier.

Dans un premier temps, elle sera caractérisée par :

- un nom de circuit : une chaîne de 30 caractères allouée dynamiquement (**char***);
- une date : une chaîne de caractères de 10 caractères (pour l'instant) selon le format 15/09/2010 (**char [11]**);
- le nombre de tours à parcourir sur le circuit en question (**int**).

On souhaite disposer des trois formes classiques de constructeurs et d'un destructeur, des méthodes classiques getXXX() et setXXX() (si pertinent) et une méthode pour afficher les caractéristiques de l'objet. Les variables de type chaîne de caractères seront donc des **char*** ou des **char[]** (le type **string** de la STL pourra être utilisé dans la 2^{ème} partie du laboratoire, c'est-à-dire à partir du 2^{ème} quart).

2.1.2 Méthodologie de développement

Un programme indépendant (**main.cxx**) testera les diverses utilisations de la classe. Celle-ci sera fournie sous forme d'une paire de fichiers **.h** et **.cxx**. On réfléchira à la méthodologie de test de manière à ce que le déroulement du programme soit compréhensible et convaincant quant au bon fonctionnement de la classe : utilisation des différents constructeurs, instanciations statiques et dynamiques (**new**), vérification du passage par le destructeur, etc.

On veillera à ce que les services proposés par la classe soient le plus possible indépendants de l'implémentation. Autrement dit, par exemple, le fait de remplacer le nom du circuit par un accès à des informations plus complexes doit impliquer le strict minimum de changements dans le code de la classe et aucune modification dans le programme à réaliser.

2.2 Associations de classes : agrégation

2.2.1 Une agrégation par valeur (ou composition)

Nous proposons d'utiliser ici, la classe **CDate** étudiée dans le cadre du cours de théorie et de remplacer, dans la classe **Race**, la chaîne de caractère par une instance d'un objet de cette classe.

On vérifiera que la modification de la classe **Race** n'implique en rien la modification des applications quiinstancieraient un objet de cette classe.

2.2.2 Une agrégation par référence

Nous allons à présent, prendre en considération un ensemble plus complet d'informations sur les circuits et créer une nouvelle classe **Circuit** qui comporte :

- un pays,
- un nom,
- une longueur exprimée en kilomètres (nombre réel positif).

En associant, par référence, un objet de type **Circuit** à notre classe **Race**, nous pouvons améliorer et compléter l'information disponible via cette classe. La méthode de **Race** qui affiche le détail d'une course, y compris le circuit sur lequel elle est effectuée devra être revue en conséquence tout comme les constructeurs de cette classe.

On pourra ensuite ajouter une méthode (inline) `getRaceLength()` qui retourne la longueur totale (exprimée en kilomètres) à parcourir par les concurrents.

On vérifiera que ce type d'agrégation permet l'utilisation du même circuit pour des courses organisées à des dates différentes.

2.2.3 Les variables membres statiques

Il existe de plus des circuits prédéfinis tels que : SPA_FRANCORCHAMPS, SILVERSTONE dont les caractéristiques bien connues sont disponibles sur les sites web ad-hoc.

SPA_FRANCORCHAMPS et SILVERSTONE sont des variables statiques de type **Circuit** de la classe **Circuit**.

2.3 Extension des classes existantes : surcharge des opérateurs

On demande de réaliser les surcharges d'opérateurs nécessaires pour qu'il soit possible de programmer les lignes de code suivantes :

1) modification ou comparaison des caractéristiques d'un circuit :

```
Circuit c, c1("Belgique","Spa_Francorchamps",7.004),  
          c2(Circuit::SILVERSTONE), c3(c1), total;  
float longueur;  
  
c = c1 ;  
c1 = c1 + 0.250 ;           // La longueur du circuit est augmentée ou  
c3 = c2 - 0.5 ;             soustraite d'une certaine distance, mesurée en  
                             km.  
longueur = c - c2;          // la longueur calculée représente la différence  
                             des longueurs des deux circuits.  
  
cout << c;  
cout << c2;  
if (c2 < c3)                // Comparaison des longueurs des circuits  
    cout << afficher message adéquat;  
total = c1 + c2 + c3 ;      // Le circuit résultant initialisé au départ à  
                             partir du circuit situé le plus à gauche des  
                             opérateurs + reçoit la longueur totale des trois  
                             circuits.
```

2) modification de la longueur du circuit par pré ou post-incrémentation. Une
incrémentation provoque une augmentation de 10 mètres de la longueur du circuit.

```
cout << ++c << endl ;  
cout << c2++ << endl ;  
cout << c2 << endl;
```


2.4 Associations de classes : héritage et agrégation

2.4.1 Héritage

Pour préparer l'utilisation de notre future application par les différents intervenants d'une compétition moteur, nous allons créer une classe **Competitor** représentant l'individu (le concurrent) qui participe à une compétition.

Dans un premier temps, nous définissons donc une classe **Competitor** décrite par son nom, son prénom, sa date de naissance, sa nationalité, son numéro de dossard et son score.

En fait, un concurrent est une personne, tout comme d'ailleurs l'administrateur de fédération ou les membres d'une équipe (constructeur). On peut donc imaginer que la classe **Competitor** dérive donc d'une classe **Person** plus générale. Celle-ci représente simplement la notion générale d'être humain qui reprendrait à la classe **Competitor**, les variables membre le représentant : nom, prénom, date de naissance, nationalité. Cette nouvelle classe possède également les méthodes élémentaires associées et des méthodes d'affichage. On redéfinira bien entendu les méthodes de la classe de base lorsque c'est nécessaire. On définira aussi un opérateur d'insertion pour la classe mère et la classe fille.

2.4.2 Hiérarchie

La classe **Person** est évidemment fort générale. En y ajoutant les classes **Competitor**, **AuthorisedPerson**, **FederationAdmin** et **TeamMember**, on obtient aisément une hiérarchie de classes adaptée à notre mise en situation. Tout l'art est bien sûr d'élaborer la hiérarchie la plus productive en termes de réutilisabilité. La classe **AuthorisedPerson** est une classe abstraite possédant une variable membre login. On peut par exemple imaginer qu'à terme, les administrateurs de fédération et les membres des équipes devront se connecter à une application au moyen d'un identifiant unique (c'est-à-dire son login) et d'un mot de passe (non mémorisé en tant que variable membre). En plus des informations disponibles dans les classes de base, les **FederationAdmin** sont caractérisés par le nom de la compétition (par exemple : FIA formule 1) qu'ils gèrent et les **TeamMember** par leur fonction et le nom de leur équipe.

A nouveau, on redéfinira bien entendu les méthodes de la classe de base lorsque c'est nécessaire, par exemple les opérateurs d'affectation ainsi que << et >>. On définira aussi une méthode virtuelle GetIdentification() qui renvoie différentes informations en fonction des cas :

Person	→	Nom
Competitor	→	PILOT : Nom#dossard#score
AuthorizedPerson	→	Impossible (classe abstraite)
FederationAdmin	→	FEDERATION : NomCompétition#Nom
TeamMember	→	TEAM : NomConstructeur#Nom#Fonction

Il est bien sûr demandé de créer tout d'abord cette petite hiérarchie puis de la tester, en particulier, grâce à un vecteur hétérogène appelant la méthode virtuelle.

2.5 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 :

- **InvalidDateException** : lancée si le format de la date encodée lors de la création d'une course ou d'une personne est valide. Celui-ci doit être le suivant : dd/mm/yyyy.
- **InvalidRaceSizeException** : lancée si la longueur calculée de la course est inférieure ou égale à 0 km.

Les classes d'exceptions proposées dérivent d'une classe de base **ExceptionBase** qui comporte la notion de message.

2.6 Les containers

2.6.1 L'utilisation future des containers

On conçoit sans peine qu'une application de gestion d'une compétition moteur va utiliser d'une part des containers mémoire divers qui permettront par exemple de représenter un ensemble de circuits disponibles, une liste de concurrents, des résultats et d'autre part des mécanismes de persistance (en clair, des fichiers). Ce second point fera l'objet du paragraphe suivant. Mettons d'abord en place une base pour nos containers.

2.6.2 Le container typique : le vecteur

Dans un tel contexte, on aura vu que le container qui revient plusieurs fois est le vecteur. Comme il sera utilisé avec divers composants, une classe **MyVector** template s'impose. Il s'agit donc :

- de créer cette classe **MyVector** template, encapsulant un tableau dynamique; il faudra définir une politique pour gérer les places libres (tassement, physique ou logique en marquant les éléments libres, à chaque disparition ou encore un vecteur de présence annexe, dont les cases contiennent 0 ou 1 selon que la case correspondante du tableau principal est libre ou pas);
- de prévoir la surcharge de l'opérateur + qui ajoute un élément dans le vecteur;
- de tester cette classe avec des **int** tout d'abord (on pense à des numéros de dossard); puis ensuite avec des objets **Race** et **Competitor**.

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

2.7 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).

2.7.1 Un fichier traceur de mouvements

Il s'agit donc de concevoir une classe **FichierLog** qui encapsule un flux et qui écrit une ligne de texte décrivant une opération réalisée dans une application en la préfixant de la date-heure système. L'utilisation ressemble à ceci :

```
FichierLog fl("RacingWorldChampionship.log");  
fl << "Login de David Coulthard (Red Bull Racing Team)";  
...  
fl << "Competitor Lewis Hamilton won at Spa-Francorchamps in 2010";
```

Le fichier résultant ressemble à ceci :

15-09-2010 14:30:15 > Login de David Coulthard (Red Bull Racing Team) 16-09-2010 14:31:04 > Competitor Lewis Hamilton won at Spa-Francorchamps in 2010

2.7.2 Un petit fichier à enregistrements

On demande de créer la classe **FichierCircuit** dont le rôle est

- ♦ d'enregistrer dans un fichier binaire dont le nom est passé au constructeur, tout objet **Circuit** passé en paramètre à sa méthode save();
- ♦ de recréer un objet **Circuit** dont on retrouve les caractéristiques à partir du nom passé en paramètre à sa méthode load();
- ♦ d'afficher son contenu sur un flux de sortie (par exemple cout) passé en paramètre à sa méthode list().

3	Enoncé du 2^{ème} quart : Développement de l'application <i>RacingWorldChampionship</i>
----------	--

<p>Date de rentrée du dossier (papier) : session de janvier 2010 Evaluation définitive de ce dossier : selon les modalités fixées par le professeur de laboratoire</p>
--

A venir ...