Correction DE 2006-2007 C++

Note : correction effectuée par moi-même pour aider à la révision. N'ayant pas assisté aux cours, il se peut que certaines réponses ne soient pas formulées de la même façon, vérifiez donc si vous souhaitez avoir les mêmes phrases que le prof.

Exercice 1

1. Une surcharge de fonction est le fait d'avoir plusieurs fonctions avec le même nom, mais avec un nombre de paramètres ou des types différents. Par exemple : int maFonction(int), double maFonction(double), int maFonction(int, int). Toutes ces fonctions sont une surcharge de maFonction.
2. Pour définir les arguments par défaut dans une fonction ont fait succédé "= valeur" pour les paramètres. Par exemple void maFonction(int toto = 5). A noter que si un paramètre a une valeur par défaut, alors toutes les suivantes doivent également avoir un paramètre par défaut. Par exemple, ceci est invalide : void maFonction (int v1 = 6, int v2) car si v1 a une valeur par défaut, toutes les variables suivantes doivent aussi avoir une variable par défaut. En revanche, ces deux prototypes sont valides : void maFonction (int v1, int v2 = 5) ou void maFonction (int v1 = 4, int v2 = 6).
3. Une référence est un "alias" vers une variable, mais contrairement à un pointeur, la référence ne peut "pointer" que vers une seule et unique variable (elle ne peut pas être modifiée). De plus, une référence ne peut pas pointer vers NULL, doit être absolument initialisée lors de sa déclaration, et enfin, ne nécessite pas le déréférencement (avec l'opérateur *) comme avec les pointeurs. Les références sont surtout utilisées lors du passage de paramètres à une fonction.

Exercice 2

1. Le constructeur sert à initialiser les attributs d'une classe (donner des valeurs par défaut, allouer de la mémoire…).
2. On peut surcharger un constructeur. Cela est utile dans de nombreux cas. Par exemple, le constructeur par copie est une surcharge (puisqu'elle porte le même nom). Mais la surcharge du constructeur ne se limite pas au constructeur de copie. Cela dépend du contexte et des besoins.
3. Les constructeurs des classes sont appelés lors de la construction d'un objet, que ce soit en utilisant le mot-clé new (donc sur le tas) ou en créant un objet sur la pile (exemple : Voiture maVoiture ; va créer un objet de type Voiture sur la pile).
4. L'opérateur new appelle tout le temps un constructeur (j'avoue ne plus être sûr, car on peut effectuer un new sur des POD – Plain Old Types – comme int, double… Et je ne sais pas si on peut parler de « constructeur » dans ce sens là. Mais je répondrais quand même oui).
5. Un destructeur ne peut pas être surchargé.

Exercice 3

Comme je comprends cet exercice, on doit ici « simuler » à la fin l'utilisation d'un Caddie, et non l'implémenter (ce qui est le sujet de l'exercice 4). Voici un exemple de l'implémentation demandée (il ne me semble pas que le sujet demande d'implémenter les fonctions, mais je vous les met quand même afin que vous compreniez bien comment fonctionne l'héritage de manière « propre »). Les explications « techniques » sont après le code.

```
class Article
{
	public:
		// Constructeur
		Article (const std::string & nomArticle_, const unsigned int prix_)
			: nomArticle (nomArticle_),
			  prix (prix_)
		{
		}

		// Destructeur virtuel pur (afin de rendre la classe abstraite)
		virtual ~Article () = 0
		{
		}

		// Renvoie le nom de l'article
		const std::string & GetNomArticle () const
		{
			return nomArticle;
		}

		// Renvoie le prix de l'article
		unsigned int GetPrix () const
		{
			return prix;
		}

	protected:
		// Nom de l'article
		std::string nomArticle;

		// Prix de l'article
		unsigned int prix;
};

class Perissable : public Article
{
	public:
		// Constructeur
		Perissable (const std::string & nomArticle_, const unsigned int prix_, const
Date & date_)
			: Article (nomArticle_, prix_),
			  datePeremption (date_)
		{
		}

	private:
		// Date de péremption
		Date datePeremption;
};

class NonPerissable : public Article
{
	public:
		// Constructeur
		NonPerissable (const std::string & nomArticle_, const unsigned int prix_)
			: Article (nomArticle_, prix_)
		{
		}
};
```

Le code est relativement simple, mais il y a quelques petites choses peut-être peu claires. Tout d'abord les listes d'initialisation. Ce sont ce qui se trouve en dessous du nom du constructeur. Par exemple, en gras:

```
// Constructeur
Article (const std::string & nomArticle_, const unsigned int prix_)
        : nomArticle (nomArticle_),
          prix (prix_)
{
}
```

Ceci est la bonne manière d'initialiser des variables membres dans un constructeur en C++. Un code peut-être plus compréhensible et qui fait la même chose :

```
// Constructeur
Article (const std::string & nomArticle_, const unsigned int prix_)
{
        nomArticle = nomArticle_ ;
        prix = prix_ ;
}
```

Toutefois, la première méthode est celle à privilégier en C++ car elle est plus rapide en terme de performances et qu'après tout… les listes d'initialisations sont faites pour ça ! A noter que ceci ne marche QUE DANS LES CONSTRUCTEURS.

Deuxième remarque : notez le mot-clé protected dans la classe Article au lieu du mot-clé private. Cela signifie que les variables membres (prix, nomArticle…) sont accessibles dans les classes filles. Autrement, en laissant private, elles ne le seraient pas.

Troisième, vous pouvez remarquez que dans les deux constructeurs des classes Perissable et NonPerissable, j'appelle le constructeur de la classe mère, en l'occurrence Article :

```
// Constructeur
NonPerissable (const std::string & nomArticle_, const unsigned int prix_)
        : Article (nomArticle_, prix_)
{
}
```

Ceci est tout à fait permis, et c'est même logique puisse NonPerissable EST UN Article, tout comme Perissable. L'intérêt de faire ceci est simple : le nom de l'article et le prix sont partagés par les deux classes, il est donc inutile de faire le même travail d'initialisation à la fois dans la classe Perissable et NonPerissable. De plus, si, un beau jour, vous décidez de changer la manière dont est copié le prix (par exemple, vous souhaitez ajouter une taxe au prix passé au constructeur), et bien vous n'avez qu'à ajouter cette taxe UNIQUEMENT dans le constructeur de la classe Article, et non dans toutes les classes filles.

Quatrième et dernière remarque : le sujet demande que la classe Article soit abstraite. Je vous rappelle que pour rendre une classe abstraite, il faut qu'elle ait au moins une fonction virtuelle pure, c'est-à-dire contenant le mot-clé virtual et se terminant par « = 0 ». Toutefois, dans mon bout d'implémentation, ni les fonctions GetNomArticle ni GetPrix n'ont vocation à être « virtualisées pures ». Une astuce très courante consiste donc à utiliser le destructeur à cette fin, ce qui explique pourquoi le destructeur est virtuel pur.

Exercice 4

J'ai utilisé la STL (std ::vector) pour cette classe, car c'est moins prise de tête que les tableaux de pointeurs… et que la STL est justement faite pour ça ! Les remarques juste après :

```
class Caddie
{
	public:
		// Destructeur
		~Caddie ()
		{
			// ICI LIBERATION DE LA MEMOIRE
		}

		// Ajoute un article dans le caddie (en utilisant la fonction push_back de la
classe std::vector)
		void AjouterArticle (const Article * article_)
		{
			articles.push_back (article_);
		}

		// Liste tous les articles du caddie (en traversant le conteneur)
		void ListerCaddie () const
		{
			for (std::size_t i = 0 ; i != articles.size() ; ++i)
				std::cout << articles[i]->GetNomArticle() << std::endl;
		}

		// Vider le caddie (en utilisant la fonction clear de la classe std::vector)
		void ViderCaddie ()
		{
			// ICI LIBERATION DE LA MEMOIRE
		}

	private:
		// Conteneur d'articles
		std::vector<Article *> articles;
};
```

Je n'ai pas fourni tout ce qui est demandé (opérateur +, opérateur <<, opérateur >>…). Tout d'abord contrairement à ce qui est dit dans le sujet, il n'est pas obligatoire ici d'avoir un constructeur. En effet, je vous rappelle que le compilateur génère automatiquement un constructeur et un destructeur par défaut si le programmeur n'en donne pas. Ici, nous n'avons rien à initialiser de « spécial » : les articles vont être ajoutés au fur et à mesure. Eventuellement, vous pouvez ajouter un constructeur de copie afin de copier le contenu d'un caddie vers l'autre.

Le destructeur et la fonction ViderCaddie doivent libérer la mémoire, ce que je n'ai pas fait ici (voir sur Internet comment libérer la mémoire des conteneurs de la STL). Si vous n'êtes pas à l'aise avec la STL, vous pouvez très bien remplacer le std ::vector par une liste chaînée « type C ».