MOOC Intro POO C++
Tutoriels semaine 2 : constructeurs/destructeurs

Les tutoriels sont des exercices qui reprennent des exemples similaires à ceux du cours et dont le corrigé est donné progressivement au fur et à mesure de la donnée de l'exercice lui-même.

Ils sont conseillés comme un premier exercice sur un sujet que l'étudiant ne pense pas encore assez maîtriser pour aborder par lui-même un exercice «classique».

Les solutions sont fournies au fur et à mesure sur les pages paires.

> Cet exercice correspond à l'exercice «*pas à pas*» page 113 de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

Le but de cet exercice est d'illustrer par un exemple détaillé, d'une part la manière d'écrire les constructeurs et destructeurs, et d'autre part de mettre en évidence leur fonctionnement.

On propose pour cela de jouer avec des animaux en peluche. Ces peluches seront caractérisées par l'*espèce* de l'animal représenté, le *nom* donné à la peluche ainsi que son *prix* de vente.

Avec vos connaissances de la semaine passée, commencez à écrire une première base pour modeliser les animaux en peluche (attributs, accesseurs, modificateur). On considèrera que les attributs `espece` et `nom` ne changent plus une fois qu'une peluche d'une espèce donnée est fabriquée (on ne la transforme pas en une autre), ce qui n'est pas le cas du prix ; on fournira donc pour ce seul attribut une méthode permettant d'en changer la valeur.

Essayez de le faire par vous même sans regarder la solution. (solution page suivante)

Solution :

```
class Peluche {
private:
    string espece;
    string nom;
    double prix;

public:
    string getEspece() const  { return espece ; }
    string getNom()    const  { return nom    ; }
    double getPrix()   const  { return prix   ; }

    void setPrix(double valeur) { prix = valeur; }
};
```

Dotons maintenant le programme d'une fonction `etiquette`, prenant une `Peluche` en argument et réalisant l'affichage (sur le terminal) de ce que pourrait être l'étiquette de cette peluche. Ceux qui objecteraient que l'affichage de l'étiquette devrait plutôt être une méthode de la classe `Peluche` ont tout à fait raison ! Mais pour des raisons pédagogiques qui seront plus claires par la suite, nous avons besoin d'une fonction « externe ».

Ajoutons également un `main()` minimaliste, qui instancie une peluche et demande l'affichage de son étiquette.

(solution page suivante)

Solution :

```
void etiquette(Peluche p) {
    cout << "Hello, mon nom est " << p.getNom() << endl
         << "Je suis un " << p.getEspece() << " et je coute "
         << p.getPrix() << " francs." << endl;
}

int main() {
    Peluche bobo;   // On utilise le constructeur par défaut
    cout << "Etiquette :" << endl;
    etiquette(bobo);
    return 0;
}
```

Le programme compile sans problème, mais donne un résultat étrange lors de l'exécution, par exemple :

```
Etiquette :
Hello, mon nom est
Je suis un  et je coute 10.3413 francs.
```

Le constructeur par défaut fourni par le compilateur s'exécute bien en tout début de programme, mais on se rend compte qu'aucun nom ni espèce n'est affiché, et surtout que le prix est fantaisiste. Cela se produit parce qu'une erreur flagrante a été commise lors de l'écriture de ce programme : le contenu de certaines variables a été utilisé sans qu'elles aient été préalablement initialisées. Le constructeur par défaut fourni par le compilateur n'initialise en effet pas les grandeurs de type de base (`int`, `double`, ...).

L'intérêt des *constructeurs* est justement d'offrir un moyen élégant pour éviter ce genre d'oublis. On peut justement remarquer que les attributs disposant d'un constructeur par défaut (dans le cas présent, les deux `string`) ont été initialisés (ici par la chaîne vide) malgré cet oubli. Ces constructeurs ont effectivement été invoqués, de la même manière que le constructeur par défaut de `Peluche` est invoqué lorsque l'on écrit :

```
Peluche bobo;
```

Donner, dans le constructeur par défaut, une valeur spécifique aux attributs (en particulier à ceux qui n'admettent pas eux-mêmes de constructeurs par défaut) serait une possibilité, mais peu satisfaisante dans le cas présent : comme l'utilisation de constantes a été choisie, il est impératif de les initialiser avec une valeur pertinente.

La solution est d'utiliser pour cela un constructeur explicite, et non pas le constructeur par défaut (qui n'a guère sa place dans un tel exemple). Les arguments de ce constructeur sont nécessairement le nom à donner à la peluche, son espèce ainsi qu'une valeur de départ pour le prix (qui est le seul élément à pouvoir être modifié ultérieurement).

À l'exécution, le programme devrait afficher :

```
[Une peluche est fabriquée]
Etiquette :
Hello, mon nom est Bobo
Je suis un ours et je coute 14.95 francs.
```

Essayez de le faire par vous même <u>sans</u> regarder la solution. (solution page suivante)

Solution :

```
class Peluche {
    // ...
    Peluche(string espece, string nom, double prixAchat)
    : espece(espece), nom(nom), prix(prixAchat)
    {
        cout << "[Une peluche est fabriquée]" << endl;
    }
    // ...
};

//...

int main() {
    Peluche bobo("ours", "Bobo", 14.95);
    cout << "Etiquette :" << endl;
    etiquette(bobo);
    return 0;
}
```

Ajoutons maintenant un *constructeur de copie* explicite (donc en remplacement de celui généré automatiquement par le compilateur -- avec ou sans attributs constants cette fois), en apportant une subtilité : au lieu de réaliser une copie parfaite, ce constructeur altérera légèrement le nom de la peluche qu'il doit copier, ce qui permettra de distinguer, lors de l'affichage de l'étiquette, les copies des originaux, par exemple :

```
[Une peluche est fabriquée]
Etiquette :
[Une peluche a été copiée]
Hello, mon nom est Bobo-copie
Je suis un ours et je coute 14.95 francs.
```

(solution page suivante)

Solution :

```
class Peluche {
    // ...
    Peluche(Peluche const& p)
    : espece(p.espece), nom(p.nom + "-copie"), prix(p.prix)
    {
        cout << "[Une peluche a été copiée]" << endl;
    }
    // ...
};
```

A noter que nous **n'**avons **pas** eu besoin de changer le `main()` pour avoir de copie !

Hé oui ! C'est bien l'étiquette d'une *copie* de la peluche `Bobo` qui est affichée. La raison est simple : la fonction `etiquette` prend une `Peluche` en paramètre, au travers d'un *passage par valeur* ; cette fonction travaille donc sur une *copie* (locale) de la peluche utilisée en argument de l'appel. C'est justement pour illustrer ce point que nous voulions une fonction externe.

Pour éviter cela, il faut bien entendu utiliser un passage *par référence*, et en l'occurrence, par une référence *constante* puisque `etiquette` n'a pas à modifier la peluche reçue en argument :

```
void etiquette(const Peluche& p) { ... }
```

À l'exécution du programme, on constate qu'il n'y a plus d'appel au constructeur de copie.

Ajoutons maintenant un *destructeur* explicite (là aussi, en remplacement de celui généré automatiquement), et modifions quelque peu les messages des constructeurs précédents de sorte qu'ils affichent le nom de la peluche concernée. Ajoutons également quelques peluches de plus dans le `main` Le programme complet est alors :

(solution page suivante)

Solution :

```
#include <string>
#include <iostream>
using namespace std;

class Peluche
{
public:
    Peluche(string espece, string nom, double prixAchat)
    : espece(espece), nom(nom), prix(prixAchat)
    { cout << "[La peluche " << nom << " est fabriquée]" << endl; }

    Peluche(Peluche const& p)
    : espece(p.espece), nom(p.nom+"-copie"), prix(p.prix)
    { cout << "[La peluche " << p.nom << " a été copiée en " << nom << "]" << endl; }

    ~Peluche()
    { cout << "[La peluche " << nom << " est cassée]" << endl; }

    string getEspece() const { return espece; }
    string getNom()    const { return nom   ; }
    double getPrix()   const { return prix  ; }

    void setPrix(double valeur) { prix = valeur; }

private:
    const string espece;
    const string nom;
    double prix;
};

void etiquette(Peluche const& p)
{
    cout << "Hello, mon nom est " << p.getNom() << endl
         << "Je suis un " << p.getEspece() << " et je coute "
         << p.getPrix() << " francs." << endl;
}

int main()
{
    Peluche bobo("ours", "Bobo", 14.95);
    cout << "Etiquette :" << endl;
    etiquette(bobo);

    Peluche* bello; // pas encore d'objet, juste un pointeur
    {
        Peluche ssss("cobra", "Ssss", 10.00);
        bello = new Peluche("toucan", "Bello", 20.00);
    }
    Peluche bello_clone(*bello);
    etiquette(bello_clone);
    delete bello;
    return 0;
}
```

Son exécution donne :

```
[La peluche Bobo est fabriquée]
Etiquette :
Hello, mon nom est Bobo
Je suis un ours et je coute 14.95 francs.
[La peluche Ssss est fabriquée]
[La peluche Bello est fabriquée]
[La peluche Ssss est cassée]
[La peluche Bello a été copiée en Bello-copie]
Hello, mon nom est Bello-copie
Je suis un toucan et je coute 20 francs.
[La peluche Bello est cassée]
[La peluche Bello-copie est cassée]
[La peluche Bobo est cassée]
```