Programmation Orientée Objet
Forme canonique
Creation d'une bibliothèque statique

Cyril Prissette
prissette@univ-tln.fr

SeaTech

Rappel des séances précédentes

Plusieurs outils du C++ assurent le bon fonctionnement d'une classe :

- L'**encapsulation** assure un contrôle d'accès sur les attributs de la classe, ce qui permet à la fois de masquer son fonctionnement interne, et -surtout- impose l'utilisation de méthodes get/set qui permettent d'effectuer un contrôle sur les valeurs des attributs.
- La **surcharge de l'operateur =** assure le bon fonctionnement de la classe en cas d'affectation. Si on ne redéfinit pas cette méthode, le C++ recopie par défaut tous les attributs des objets. Ceci suffit en général mais peut être source d'erreur dans certains cas, comme par exemple avec une classe qui manipulerait des pointeurs.

Aujourd'hui, on va élargir le cas de l'affectation à d'autres moments clés dans le cycle de vie d'un objet qui peuvent poser problème.
Puis, on verra comment organiser le code source d'une classe, afin d'en faire une bibliothèque et de pouvoir compiler facilement cette bibliothèque et des programmes qui y font appel.

Constructeurs

Vous avez écrit une méthode initialiser(). Elle fonctionne, mais ce n'est pas tout à fait satisfaisant :

- il faut l'appeler explicitement pour chaque objet qu'on instancie, ce qui rend la rédaction du programme assez lourde et ce qui est source d'erreur si on oublie un appel à initialiser().
- on ne peut pas s'en sevir pour donner des valeurs spécifiques aux attributs au moment où l'objet est créé.
- lors d'un appel de fonction/méthode, des objets peuvent être instanciés en tant que paramètre de la fonction/méthode et on est pas tout à fait sûr de ce qui se passe. Même si l'opérateur = est utilisé implicitement pour copier les valeurs des paramètres[1], ce n'est pas forcement suffisant pour initialiser les objets.

[1] ce qui n'est pas le cas, mais admettons...

Constructeurs

Pour résoudre tous ces problèmes, le C++ offre un mécanisme particulier : il est possible de définir des méthodes d'initialisation qui seront automatiquement appelées lors de l'instanciation d'un nouvel objet. Ces méthodes sont les **constructeurs** de la classe.

Les constructeurs sont des méthodes, avec deux particularité syntaxiques :

- ils ont le même nom que la classe.
- ils n'ont pas de type de retour, même pas `void`.

Constructeur par défaut

Quand on instancie un objet d'une classe, sans plus de précision, c'est le **constructeur par défaut** qui est appelé. Pour l'écrire, il suffit de définir un constructeur sans aucun paramètre.
Modifiez votre programme pour que la méthode :

```
void Vehicule::initialiser(void)
```

devienne :

```
Vehicule::Vehicule()
```

Pour visualiser son appel, ajoutez un affichage dans cette méthode :

```
cerr << "constructeur par defaut de Vehicule" << endl;
```

★

Constructeurs parametrés

Souvent, juste après avoir instancié un objet d'une classe, on modifie ses attributs.

```
Vehicule moto;
moto.set_vitesse(60);
```

Ce serait plus *user-friendly* si on pouvait écrire :

```
Vehicule harley(60); // mettre immédiatement la vitess
Vehicule suzuki(60,240); // définir aussi la vitesse m
Vehicule ducati(60,240,0.5); // définir aussi le poids
```

Commençons par le cas de la harley, avec un seul paramètre. Dupliquez votre constructeur par défaut afin de lui ajouter un paramètre pour la vitesse. Pour visualiser son appel, ajoutez un affichage dans cette méthode

```
cerr << "cstror parametre de Vehicule (1)" << endl;
```

★

Constructeurs parametrés

Faites de même pour les constructeurs parametrés avec 2 et 3 paramètres, en ajoutant dans ces 2 méthodes un affichage spécifique :

```
cerr << "cstror parametre de Vehicule (2)" << endl;
```

ou

```
cerr << "cstror parametre de Vehicule (3)" << endl;
```

Faites fonctionner le main() suivant :

```
int main()
{
Vehicule harley(60); // modifie vitesse
Vehicule suzuki(100,240); // modifie aussi vitesse max
Vehicule ducati(120,240,0.5); // modifie aussi le poids
}
```

★

Constructeurs parametrés

Executez le main() suivant :

```
int main()
{
Vehicule honda=100;
}
```

Cette syntaxe fait-elle appel à l'opérateur d'affectation ou à un constructeur parametré? Pour en être sûr, ajoutez l'affichage d'un message à la méthode operator=().

★

Constructeurs par recopie

Executez maintenant le programme suivant :

```
double energie_cinetique(Vehicule V)
{
return V.poids * V.get_vitesse() * V.get_vitesse();
}

int main()
{
Vehicule yamaha=70;
cout << "Ec=" << energie_cinetique(yamaha);

}
```

★ conservez l'affichage du programme, il y a des choses à voir.

Constructeurs par recopie

La fonction `energie_cinetique()` instancie un Vehicule nommé V qu'elle utilise comme paramètre. Pourtant, l'affichage n'indique pas qu'un constructeur est appelé.
De même, les attributs de l'objet yamaha sont copiés dans V le temps d'évaluer la fonction. Mais l'affichage n'indique pas d'appel à l'opérateur d'affectation pour effectuer V=yamaha.

L'explication est la suivante : quand un **objet est construit en dupliquant un autre objet** de la même classe, un constructeur spécial est appelé. Il s'agit du **constructeur par recopie**.

Dans le programme précédent, le C++ a fait appel au constructeur par recopie générique, qui se contente de recopier les attributs.

Constructeurs par recopie

Le constructeur par recopie est un constructeur parametré, dont le paramètre est un objet de la classe. On pourrait *a priori* l'écrire :

```
Vehicule::Vehicule(Vehicule V)
{
cerr << "cstror recopie de Vehicule" << endl;
vitesse=V.vitesse;  // ou vitesses[0], selon votre version
vitesse_max=V.vitesse_max; // ou vitesses[1]
poids=V.poids;
}
```

Essayez de définir ce constructeur.

★ ça devrait être refusé à la compilation, ou au pire bloquer l'execution du programme.

Constructeurs par recopie

Le problème d'écrire `Vehicule::Vehicule(Vehicule V)` c'est qu'il s'agit d'une méthode dont un paramètre est un Véhicule.
Donc l'execution de cette méthode va appeler le constructeur par recopie de Vehicule pour recopier le paramètre Vehicule, c'est à dire que la méthode s'appelle elle-même recursivement. Sans borne de récursivité, ça ne peut que planter.

La solution est donc de ne pas passer une copie du Vehicule, mais le Vehicule lui même . Vous avez déjà rencontré cette situation (mais pas aujourd'hui).
Trouvez un moyen de faire fonctionner le constructeur par recopie.
★

Destructeur

De la même façon que les constructeurs s'exécutent automatiquement au début de la vie des objets, il est possible de définir une méthode qui s'execute automatiquement à la fin de la vie des objets : le **destructeur**.

Le prototype du destructeur est le même que celui du constructeur par défaut, à une différence prêt : on ajoute un ~ devant le nom de la méthode.

Pour une raison qu'on verra plus tard dans le cours, il est généralement préférable de déclarer le destructeur en `virtual`. Pour l'instant, peu importe. On y reviendra en temps voulu.

Ecrire le destructeur de la classe Vehicule, en lui faisant afficher un message quand il s'execute. Comme notre classe Vehicule n'est pas très complexe, le destructeur n'a rien d'autre à faire.

★

Destructeur

De la même façon que les constructeurs s'exécutent automatiquement au début de la vie des objets, il est possible de définir une méthode qui s'execute automatiquement à la fin de la vie des objets : le **destructeur**.

Le prototype du destructeur est le même que celui du constructeur par défaut, à une différence prêt : on ajoute un ~ devant le nom de la méthode.

Pour une raison qu'on verra plus tard dans le cours, il est généralement préférable de déclarer le destructeur en `virtual`. Pour l'instant, peu importe. On y reviendra en temps voulu.

Ecrire le destructeur de la classe Vehicule, en lui faisant afficher un message quand il s'execute. Comme notre classe Vehicule n'est pas très complexe, le destructeur n'a rien d'autre à faire.

★

Forme canonique

Il est temps de faire le point.

Quand on manipule une classe complexe, comme une classe utilisant des pointeurs, le cycle de vie des objets présente 4 moments critiques qui sont sources de nombreux bugs particulièrement pénibles à corriger : utilisation de zone mémoire non allouée avec des crashs intempestifs, création involontaire de plusieurs objets partageant les mêmes données, fuite de mémoire..

Ces 4 moments critiques sont :

- la creation de l'objet
- l'affection
- l'utilisation d'un objet comme paramètre de fonction
- la disparition de l'objet

Forme canonique

Ces 4 moments critiques sont respectivement couverts par :

- **le constructeur par défaut**
- **l'opérateur d'affectation**
- **le constructeur par recopie**
- **le destructeur**

De même qu'on utilisera systématiquement l'encapsulation pour assurer l'intégrité des données, on s'imposera désormais d'écrire les classes sous **forme canonique** en définissant systématiquement ces **4 méthodes**.

Classe Fraction

Ecrire la classe Fraction. Cette classe doit avoir 2 attributs entiers nommés *numerateur* et denominateur Elle doit être encapsulée et sous forme canonique.
Elle doit permettre de faire fonctionner le main() suivant :

```
int main()
{
Fraction A;       // vaut 0/1
Fraction B(1,2); // vaut 1/2
Fraction C=3;     // vaut 3/1
A=B+C;
cout << A << endl;
}
```

★

Bibliothèque

Imaginons qu'on souhaite ré-utiliser la classe Fraction dans un autre programme. Il faudrait pouvoir importer la classe et le fonctionnement de ses méthodes. Il s'agit essentiellement d'organiser son code source, pour simplifier sa ré-utilisation.
Pour cela, on éclate le programme entre plusieurs fichiers :

- un fichier fraction.h qui contient la définition de la classe
- un fichier fraction.cpp qui contient le corps des méthodes
- un fichier .cpp qui contient la fonction main(). Ce fichier s'appelle souvent main.cpp par défaut, mais je préfère l'appeler test_fraction.cpp
- un fichier makefile qui contient les directives de compilation

Bibliothèque

fraction.h

Le fichier .h est le fichier d'entête qui sert au compilateur à faire une première analyse syntaxique, afin de vérifier la cohérence du programme. C'est à dire si il connait toutes les classes, méthodes, fonctions.. Si les fonctions et les méthodes ont le bon nombre de paramètre et les types sont cohérents, etc.
Ce fichier contient tous les *#include* nécessaires à la classe, suivis de la déclaration de la classe.
Enfin, on rajoute généralement un garde-fou grâce à la directive *#ifndef* pour éviter tout problème si quelqu'un écrivait :

```
#include "fraction.h"
#include "fraction.h"
```

Bibliothèque

fraction.h

```
#ifndef __FRACTION_H
#define __FRACTION_H

#include <iostream>
using namesapce std;

class Fraction
{
...

};

#endif
```

Bibliothèque

fraction.cpp

Le fichier fraction.cpp contient le corps des méthodes et des fonctions amies. Ce fichier, une fois compilé, formera le coeur de la bibliothèque. Il inclut le fichier fraction.h, afin d'avoir accès à la définition de la classe. Attention : ne surtout pas définir ici la fonction main().

Bibliothèque

fraction.cpp

```
#include "fraction.h"


Fraction::Fraction()
{
numerateur=0;
denominateur=1;
}

Fraction::Fraction(const Fraction& f)
{
numerateur=f.numerateur;

...
```

Bibliothèque

test_fraction.cpp

Le fichier test_fraction.cpp contient une fonction main().
Pour commencer, il suffit de fournir un exemple d'utilisation des différentes méthodes, pour vérifier le bon fonctionnement de l'ensemble de la classe.

Bibliothèque

test_fraction.cpp

```
#include "fraction.h"

int main()
{
Fraction A;       // vaut 0/1
Fraction B(1,2); // vaut 1/2
Fraction C=3;     // vaut 3/1
A=B+C;
cout << A << endl;
}
```

Bibliothèque

makefile

Le fichier makefile est utilisé par l'utilitaire make, pour automatiser la compilation du programme et de la/les bibliothèques qu'il utilise.
L'utilitaire make se charge de vérifier quels fichiers necessitent d'être recompilés dans l'ensemble du projet.
C'est un fichier texte qui contient une suite d'instructions sous la forme :

```
fichier_a_creer : fichiers_necessaires
    commande_de_compilation
```

Bibliothèque

makefile

```
test_fraction : test_fraction.cpp fraction.o
    g++ -o test_fraction test_fraction.cpp fraction.o

fraction.o : fraction.cpp fraction.h
    g++ -c fraction.cpp
```

Bibliothèque

Complétez les différents fichiers relatifs à la classe Fraction, afin d'en faire une bibliothèque. Compilez le programme de test dans un terminal, à l'aide de la commande :

```
make test_fraction
```