Programmation Orientée Objet
Surcharges

Cyril Prissette
prissette@univ-tln.fr

SeaTech

Surcharge d'opérateurs

Comme on l'a vu au premier cours avec `Vehicule attacher(Vehicule)`, puis`Vehicule operator+(Vehicule)`, il est possible de definir le comportement de operateur d'addition + pour une classe : il s'agit simplement de définir la méthode `operator+()` pour cette classe.

De manière générale, pour n'importe quel opérateur `X`, le compilateur interprète l'instruction

```
a X b
```

comme :

```
a.operatorX(b)
```

Surcharge d'opérateurs

Commençons de manière simple par écrire une surcharge de l'opérateur * permettant de faire fonctionner le main() suivant :

```
int main()
{
Vehicule moto;
moto.initialiser();
moto.set_vitesse(60);

moto = moto * 1.5; // passe la vitesse a 90
moto.afficher();

moto = moto * 1.5; // passe a vitesse_max (au lieu de 135)
moto.afficher();
}
```

★ Au travail!

Correction - Vehicule Vehicule::operator*(double)

Si on utilise la fonction `set_vitesse()`, on n'a pas à se soucier de limiter la vitesse a vitesse_max, puisque `set_vitesse()` le fait.
Il est toujours préférable de profiter des fonctions déjà écrites.

```
Vehicule Vehicule::operator*(double facteur)
{
Vehicule V;
V.set_vitesse( vitesse*facteur );
V.vitesse_max = vitesse_max;
V.poids = poids;
return V;
}
```

Correction - Vehicule Vehicule::operator*(double)

Remarque : dans la méthode précédente, l'affection des attributs vitesse_max et poids peut être écrite de manière plus concise, en affectant directement à V une copie de l'objet qui exécute la méthode `operator*()`
Pour cela, on peut utiliser le fait que chaque objet connaisse sa propre adresse mémoire. Cette adresse lui est donnée par le pointeur **this**
Un objet peut donc exprimer sa propre valeur en déréférençant ce pointeur, c'est à dire avec ***this**. On peut aussi dire que pour un objet, ***this** signifie "lui-même".
Utiliser cette syntaxe pour ré-écrire la méthode, afin de copier d'un seul coup les valeurs de vitesse, vitesses_max et poids, avant d'effectuer `V.set_vitesse( vitesse*facteur );`
★

Correction - Vehicule Vehicule::operator*(double)

```
Vehicule Vehicule::operator*(double facteur)
{
Vehicule V = *this;  // V est une copie de moi-même
V.set_vitesse( vitesse*facteur );
return V;
}
```

(Fin de l'apparté sur ***this**)

Surcharge d'opérateurs

A présent, on souhaite écrire une surcharge de l'opérateur * permettant de multiplier la vitesse du Vehicule par un double. Si le resultat doit dépasser vitesse_max, ne pas modifier la vitesse.

```
int main()
{
Vehicule moto;
moto.initialiser();
moto.set_vitesse(60);

moto = 1.5 * moto;
moto.afficher(); // 90

moto = 1.5 * moto;
moto.afficher(); // 135 interdit, on reste a 90
}
```

Surcharge d'opérateurs

Problème :

```
1.5 * moto
```

Dans ce cas, le compilateur devrait comprendre

```
(1.5).operator*(moto)
```

Mais l'opérande de gauche (c'est à dire 1.5) n'est pas un objet de la classe Vehicule.

Donc on ne peut pas définir cet operator* comme étant une méthode de la classe Vehicule.

Surcharge d'opérateurs

Solution :
Le compilateur peut (aussi) interprêter `1.5 * moto` comme un appel de fonction (attention *fonction* $\neq$ *methode*) :

```
operator*(1.5, moto)
```

Il suffit de définir la fonction "en C"

```
Vehicule operator*(double, Vehicule)
```

Ce n'est pas aussi élégant qu'avec une méthode, mais quand l'operande de gauche n'est pas un objet de la classe que l'on définit, on ne peut pas faire autrement.

Ecrire en dehors de la classe la fonction
`Vehicule operator*(double, Vehicule)`
★

Surcharge d'opérateurs

Après la définition de la classe, et avant la fonction main() :

```
Vehicule operator*(double f, Vehicule Veh)
{
Vehicule V;
V=Veh;
V.set_vitesse( f * Veh.get_vitesse() );
return V;
}
```

Fonction amie

La fonction `Vehicule operator*(double f, Vehicule Veh)` n'est pas un membre de le classe Vehicule. Elle est donc soumise à l'encapsulation, ce qui rend son écriture un peu lourde.
Il est possible de prévoir dans la classe qu'une fonction (attention *fonction* $\neq$ *methode*) aura le droit d'accéder aux protected. Pour cela, la fonction doit être prototypée dans la classe et y être précédée du mot clé `friend`.
Transformer la fonction `Vehicule operator*(double, Vehicule)` pour qu'elle soit reconnue comme fonction amie par la classe Vehicule, afin de pouvoir écrire simplement Veh.vitesse au lieu de Veh.get_vitesse().
★

Surcharge d'opérateurs - fonction amie - friend

```
class Vehicule
{
...
friend Vehicule operator*(double, Vehicule);
};

Vehicule operator*(double f, Vehicule Veh)
{
Vehicule V;
V=Veh;
V.set_vitesse( f * Veh.vitesse );
return V;
}
```

C'est très utile quand l'opérande de gauche d'un opérateur n'est pas un objet de la classe qu'on programme. Le reste du temps, c'est à éviter absolument.

Surcharge d'opérateurs - operator=

De même que pour les opérateurs + ou *, on peut redéfinir l'opérateur =
C'est utile pour éviter l'affection par défaut qui consiste à recopier tous les attributs et qui n'est pas toujours souhaitable, par exemple si la classe manipule des pointeurs.
Cet opérateur est *a priori* assez facile à écrire, si on se contente de voir `voiture1=voiture2` comme `voiture1.operator=(voiture2)`
Ecrire la méthode de surcharge de l'affection, c'est à dire de l'operator=
★

Surcharge d'opérateurs - operator=

```
void Vehicule::operator=(Vehicule V)
{
// en tant qu'objet, je dois ressembler à V
vitesse=V.vitesse;
vitesse_max=V.vitesse_max;
poids=V.poids;
}
```

Surcharge d'opérateurs - operator=

Cette première version de l'affection marche, mais elle n'est pas parfaite. En effet, en C++, on doit pouvoir écrire :

```
a = b = c = d;
```

Qui s'execute comme une suite d'affectations, résolues de droite à gauche :

```
(a =(b =(c = d))); // priorité à droite
```

Pour que les calculs s'enchainent correctement, il faut qu'à la fin de l'affection, on renvoie la valeur affectée.

Corriger la méthode, afin qu'elle permette cette syntaxe. Elle devient donc `Vehicule Vehicule::operator=(Vehicule V)`

★

Surcharge d'opérateurs - operator=

```
Vehicule Vehicule::operator=(Vehicule V)
{
vitesse=V.vitesse;
vitesse_max=V.vitesse_max;
poids=V.poids;

return *this;
}
```

Remarque : on pourrait faire un `return V;`, mais la pratique conseille d'utiliser plutôt la valeur de l'objet qui a appelé la méthode.

Surcharge d'opérateurs - operator=

L'opérateur est encore perfectible. En effet, il ne gère pas les cas où on écrirait :

```
moto=moto;
```

Pour l'instant, on recopie les attributs de Vehicule, ce qui est au mieux inutile, et qui peut être source de bugs (si la classe utilise des pointeurs, par exemple). Il serait préférable de ne rien faire dans ce cas.

Pour cela, il faut pouvoir comparer les adresses mémoires de l'objet qui appelle la méthode, et de l'objet passé en paramètre (ce qui implique que le paramètre doit être un référence &).

Surcharge d'opérateurs - operator=

Pour cela, on devrait aussi rendre non modifiable (const) le paramêtre de la fonction : on souhaite connaitre son adresse mémoire, mais on ne doit pas être autorisé modifier le paramêtre.
Enfin, pour rendre les notations cohérentes entre elles, le type de retour devrait également être une référence.

Corriger la méthode afin qu'elle evite l'auto-affectation. Elle deviendra pour cela `Vehicule& Vehicule::operator=(const Vehicule &V)`
★

Surcharge d'opérateurs - operator= - correction

```
// Retour : renvoie par référence
// Parametre : passage par reference constante
Vehicule& Vehicule::operator=(const Vehicule& V)
{
if (&V==this) // si on essaye de faire voiture=voiture;
        return *this; // interrompre la methode

vitesse = V.vitesse;
vitesse_max = V.vitesse_max;
poids = V.poids;

return *this; // renvoie de l'objet lui-même
}
```

Surcharge d'opérateurs - operator<<

Le dernier opérateur à savoir écrire est l'opérateur d'affichage <<, il regroupe à lui seul plusieurs notions vues aujourd'hui.
On souhaite pouvoir écrire :

```
cout << moto << endl;
```

Au lieu de :

```
moto.affichage();
```

Le résultat à l'écran doit rester le même.

Surcharge d'opérateurs - operator<<

Voici les indices pour vous guider :
1) ici l'affichage a lieu sur le `cout`, qui est de type `ostream&` (c'est à dire **o**utput **stream**, ou en français "flux de sortie").
A vrai dire, l'opérateur devrait marcher pour n'importe quel ostream& et pas uniquement cout.
2) comme vous l'aurez remarqué, l'opérande de gauche n'est pas un objet de la classe Vehicule. Vous avez déjà rencontré cette situation quand on a voulu écrire `1.5 * moto`
3) il faut pouvoir enchainer les operateurs d'affichages <<. vous avez déjà rencontré cette situation avec l'opérateur =
★

Surcharge d'opérateurs - operator<<

```
class Vehicule
{
...
friend ostream& operator<<(ostream& c, Vehicule v) // cf 2)
};


ostream& operator<<(ostream& c, Vehicule v)
{
c << vitesse << "/" << vitesse_max << "," << poids; // cf 1)

return c; // cf 3)
}
```

Surcharge d'opérateurs - operator>>

En bonus : l'opérateur >>

```
class Vehicule
{
...
friend istream& operator>>(istream& c, Vehicule& v)
};


istream& operator>>(istream& c, Vehicule& v)
{
c >> vitesse >> vitesse_max >> poids;

return c;
}
```