Chapitre 4 : classes, surcharge de fonctions

1 Classes

1.1 Principe

Les *classes* constituent le principal apport du `C++` par rapport au `C`. L'idée est d'introduire une syntaxe permettant de manipuler plusieurs variables sous un même nom.

Prenons l'exemple suivant : on souhaite effectuer des calculs sur les nombres complexes. Un nombre complexe peut être défini par la donnée de ses parties réelles et imaginaires. Il serait donc possible de manipuler des nombres complexes en déclarant des variables de type `double` deux par deux, par exemple `double x=1, y=3` pour le nombre $1+3i$. Pour calculer, par exemple, le conjugué d'un nombre complexe, on utiliserait une fonction de prototype `void conj(double x, double y, double& conj_x, double& conj_y)` telle que `conj(1,-1,x,y)` affecterait à `x` la partie réelle de $\overline{1-i}$ et à `y` sa partie imaginaire. Cette syntaxe va vite s'avérer très lourde.

Il est en fait possible de définir un nouveau type, `complexe` par exemple, avec la syntaxe

```
class complexe{
double x, y;
};
```

On a déclaré une "classe", qui se manipulera comme un nouveau type. Remarquer que la déclaration d'une classe doit être suivie d'un point-virgule, ce qui est une erreur fréquente pas forcément simple à repérer.

Comme pour n'importe quel type, on pourra déclarer des variables de type `complexe` avec "`complexe z;`". La variable `z` dispose alors de deux champs `x` et `y`, de type `double`, qui sont accessibles par `z.x` et `z.y`. On peut alors calculer le conjugué d'un complexe par une fonction de prototype `complexe conj(complexe)`, ce qui correspond beaucoup plus à l'intuition.

Ici, les champs `x` et `y` seront appelés des *membres* de la classe `complexe`. Les variables de types `complexe` seront appelées des *instances* de cette classe.

Un des intérêts de la notion de classe est qu'il est possible de définir comme membre de la classe non seulement des variables ayant des types de bases, mais aussi des fonctions. Les fonctions membres seront appelées des *méthodes*. Par exemple, il est naturel ici de définir le calcul du conjugué comme une méthode de la classe `complexe`. On accèdera au conjugué d'une variable `z` par la syntaxe `z.conj()`.

1.2 Membres privés et publics

Les membres d'une classe peuvent être soit *privés*, indiqués par le mot-clé `private`, soit *publics*, indiqués par le mot-clé `public`.

Les membres publics peuvent être appelés par n'importe quelle fonction, en revanche les membres privés ne peuvent être appelés que par les méthodes de la classe et par certaines fonctions dites fonctions "amies" qui auront explicitement été déclarées comme telles dans la classe.

L'intérêt de cette restriction est de permettre de vérifier que les modifications des champs privés sont faites en respectant certaines règles. Par exemple, si on définit une classe implémentant les rationnels par la donnée de leur numérateur et de leur dénominateur, on peut vouloir que les fractions obtenues soient toujours irréductibles. Rendre privés le numérateur et le dénominateur permet de s'assurer que cela sera toujours le cas : en effet, les modifications du numérateur et du dénominateur n'auront plus lieu que dans le cadre des méthodes de la classe, et il suffira de s'assurer que ces dernières renvoient bien des résultats sous forme réduite pour savoir que tous les rationnels manipulés seront également dans ce cas.

1.3 Constructeur, destructeur

Toute classe va contenir plusieurs méthodes particulières :

— Les *constructeurs* sont des méthodes qui permettent de créer une instance de la classe à partir de certains paramètres. Ces méthodes ont le même nom que la classe correspondante.
Un constructeur est appelé implicitement à chaque déclaration de variable. De même, l'appel de `complexe(arg)` va correspondre à un appel de constructeur.
Il est conseillé qu'au moins un des constructeurs n'ait pas d'argument (on parlera alors de constructeur par défaut), de sorte à pouvoir écrire simplement "`ma_classe x`". La variable `x` a ici été créée en appelant le constructeur par défaut.
Pour la classe `complexe`, on peut penser à un constructeur qui définit un complexe à partir de sa partie réelle et de sa partie imaginaire, ou encore un constructeur qui crée le nombres complexe x à partir du réel x.
Il existera aussi un constructeur, appelé constructeur *par copie*, qui créera, à partir d'une instance de la classe, une deuxième instance correspondant aux mêmes valeurs. Le constructeur par copie aura donc un prototype de la forme suivante :
`ma_classe(ma_classe const&);`
(il prend en argument une référence sur un objet constant et n'a pas de type retour).

— Le destructeur est une méthode qui est appelé automatiquement à chaque fois qu'une instance de classe est effacée, par exemple lorsque l'on quitte une fonction dans laquelle une instance locale avait été déclarée.
Dans une classe `ma_classe`, le destructeur a pour nom `~ma_classe` et a pour prototype
`~ma_classe (void);`
(pas d'arguments, et pas de type de retour).

— L'opérateur d'affectation, ou opérateur `=` permet de copier le contenu d'une instance dans une autre instance. Il a le prototype
`ma_classe& operator=(ma_classe const&);`
(prend en argument une référence constante sur un objet de la classe et renvoie une référence sur un objet de la classe).

Si l'on a pas définit de constructeur par copie, de destructeur et d'opérateur d'affectation, ces méthodes sont créées automatiquement.

Toutefois, dans certains cas, il est naturel que l'un des champs de la classe soit un pointeur vers une zone mémoire ne faisant pas partie de la classe, ce qui va poser des problèmes dans ces méthodes. Pour initialiser, détruire et copier correctement les instances de ces classes, il faut redéfinir à la main un constructeur par copie, un destructeur et un opérateur d'affectation.

2 Surcharge de fonctions et d'opérateurs

2.1 Surcharge de fonctions

En C++, il est possible de donner le même nom à plusieurs fonctions. Par exemple, on peut définir une fonction `somme` qui prend en argument deux variables de type `double` et qui renvoie leur somme. La fonction `somme` a donc pour prototype `double somme(double, double);`. On voudrait aussi définir une fonction ayant le même nom et le même effet sur les `int` ou les `complexe`. Il est possible de le faire, la seule contrainte étant qu'il ne doit pas y avoir d'ambiguïté sur l'identité de la fonction appelée. Autrement dit, deux fonctions ayant le même noms doivent nécessairement avoir des types d'arguments différents.

2.2 Arguments par défaut

Il est également possible de définir des arguments par défaut pour une fonction. Il s'agit en fait simplement d'une manière de surcharger automatiquement des fonctions. Par exemple, le prototype `double mafonction(int n, int m=2, double x=0.);` est en fait équivalent aux *trois* prototypes suivants :

- `double mafonction(int n);`
- `double mafonction(int n, int m);`
- `double mafonction(int n, int m, double x);`

Si `n` et `m` sont des variables de type `int`, les appels `mafonction(n)` et `mafonction(n,m)` seront respectivement équivalents aux appels `mafonction(n,2,0.)` et `mafonction(n,m,0.)`.

2.3 Surcharge d'opérateurs

La plupart des opérateurs du C++ (par exemple, les opérateurs `+`, `*`, `-`, `/`, etc.) sont en fait des fonctions qui peuvent donc être surchargées. Un opérateur peut également être surchargé en tant que méthode d'une classe.

Dans l'exemple d'une classe représentant les nombres complexes, cela permet par exemple d'utiliser une syntaxe comme `z1 + z2` ou `z1 * z2` pour calculer la somme et le produit de deux nombres complexes, plutôt que de passer par des fonctions, comme par exemple `somme(z1,z1)` ou `produit(z1,z2)`.

Les prototypes "classiques" des opérateurs surchargés sont résumés dans les tableaux ci-dessous. À noter :

- On déclare au maximum les objets comme *constants*, pour éviter des erreurs dues à une modification d'un objet qui aurait dû rester inchangé.
- On utilise au maximum des références, pour éviter des recopies d'objets qui pourraient être coûteuses (par exemple si les objets considérés manipulent des tableaux de grande taille).

Le tableau suivant liste les principaux opérateurs qui sont naturellement surchargés par des méthodes.

Opérateur	Prototype	Appel	Signification
`=`	`ma_classe& operator= (ma_classe const&);`	`x=y`	`x.operator=(y)`
`+,-`	`ma_classe operator- (void) const;`	`-x`	`x.operator-()`
`+=,*=,/=,...`	`ma_classe& operator+=(ma_classe const&);`	`x+=y`	`x.operator+=(y)`
`++,--`	`ma_classe& operator++(void);`	`++x`	`x.operator++()`
`++,--`	`ma_classe operator++(int);`	`x++`	`x.operator++(0)`
`(),[]`	`type_retour operator() (type_args) const;`	`x(a,b)`	`x.operator()(a,b)`

Surcharge par des méthodes de la classe `ma_classe`

Le tableau suivant liste les principaux opérateurs qui sont naturellement surchargés par des fonctions amies.

Opérateur	Prototype	Appel	Signification
+,*,/,...	`ma_classe operator+ (ma_classe const&, ma_classe const&);`	`x+y`	`operator+(x,y)`
==,!=,<,...	`bool operator==(ma_classe const&, ma_classe const&);`	`x==y`	`operator==(x,y)`
<<	`std::ostream& operator<<(std::ostream &, ma_classe const&);`	`o << x`	`operator<<(o,x)`

Surcharge par des fonctions amies de la classe `ma_classe`

Note : le mot-clé `const` après le prototype d'une méthode signifie que l'objet attaché à la méthode appelée ne peut pas être modifié, sous peine d'une erreur à la compilation.