Surcharge de sélection

1. Méthodes de classes

Contrairement aux autres langages de programmation où la portée d'un identificateur est unique pour une application donnée, une et une seule fonction ou routine peut s'appeler **toto** pour tout un programme, en C++ la portée d'un identificateur dépend du contexte d'appel.

Imaginons un autre type d'objet géométrique, le rectangle, qui disposerait des mêmes méthodes que le cercle :

```
class Rectangle
{
      // Methodes utilisateurs public:
void                    Move(float deltax, float deltay);
void                Zoom(float scale);
float Area();
etc.
```

Dans le code utilisateur suivant :

```
Cercle c1;
Rectangle r1;
c1.Zoom(3.5);
r1.Zoom(4.0);
```

le compilateur lève les ambiguïtés de nom selon le contexte d'appel, l'écriture **c1. Zoom** désigne la méthode **Zoom** de la classe **Cercle**, appliquée à l'objet **c1**, alors que **r1. Zoom** désigne la méthode **Zoom** de la classe **Rectangle**, appliquée à **r1**.

La syntaxe d'implémentation des méthodes de classes, est d'ailleurs parlante, les points d'entrée sont **Cercle: : Zoom** ou **Rectangle :: Zoom**.

2. Surcharges de sélection

Un second mécanisme, très puissant, permet d'implanter dans une même classe plusieurs méthodes de même nom, dès l'instant que la liste d'arguments d'appel, nombre et nature, peut lever l'ambiguïté.

Par exemple :

```
class Machin {
void Calcul(int); // Interface 1 entier, Ci
void Calcul(int, int); // Interface 2 entiers, Cii
void Calcul(float, float); // Interface 2 reels, Cff ...
```

Lors de l'utilisation, c'est le compilateur qui va choisir l'appel correct en fonction du contexte :

```
int n;
float x, y; Machin M;
M.Calcul(n, 3); // Appel Cii
M.Calcul(x, y); // Appel Cff
M.Calcul(x, 1); // Appel Cff, conversion 1 -> 1.0
M.Calcul(x, n); // Appel Cff avec cast (float)n
M.Calcul(5); // Appel Ci
M.Calcul(x); // ERREUR !
M.Calcul((int)x); // Cast explicite, appel Ci
```

Contrairement à d'autres langages où la même fonction peut exister sous tout un tas de noms différents selon les types d'arguments (voir en Fortran les **sin**, **dsin**, **imod**, **jmod**, **amod**, etc.), en C++, et sous réserve d'avoir prévu différentes configurations opératoires, on invoque tel ou tel traitement sous une appellation fonctionnelle générique en laissant au compilateur le soin de traiter les détails d'intendance !

Ce mécanisme a déjà été utilisé, sans explications, dans le cas du cercle et ses deux constructeurs, **Cercle()** et **Cercle (float)**. C'est exactement une mise en œuvre de ce mécanisme de sélection.

3. Arguments optionnels

Un dernier mécanisme permet de déclarer des méthodes avec des arguments optionnels et valeurs par défaut s'ils ne sont pas fournis lors de l'appel.

Dans l'exemple du cercle, on aurait pu n'implanter qu'un seul constructeur, celui avec un argument rayon optionnel. Si l'argument n'est pas fourni à l'appel, le compilateur utilisera la valeur par défaut.

L'interface est la suivante :

```
class Cercle
{
Cercle(float rayon = 1.0);
...
```

et l'implémentation ne comporte qu'un constructeur :

```
/* Constructeur avec rayon */
Cercle::Cercle(float rayon)
{
cx = cY = 0.0;  // A l'origine par défaut
cR = rayon; // Rayon utilisateur
}
```

A l'utilisation, le compilateur complètera l'appel automatiquement :

```
Cercle c1(5.0); // Rayon initial 5.0 explicite
Cercle c2; // c2(1.0) implicite
```

Dans cet exemple, l'intérêt est une simplification de l'implémentation, une seule méthode au lieu de deux.

- Un autre type d'utilisation, très utile en pratique, concerne l'évolution de développements C++. Supposons qu'un jour la librairie d'objets géométriques, **Cercle**, **Rectangle**, etc., évolue vers une version 3D.

  La classe **Cercle** devra être revue, avec trois attributs coordonnés du centre, **cx**, **cy**, **cz**. La méthode effectuant un déplacement relatif deviendra :

```
void Move(float dx, float dy, float dz);
```

  Doit-on remettre à hauteur toutes les applications 2D existantes ? Pas nécessairement, on peut décider que les anciennes applications 2D travailleront dans le plan z = 0; il suffit alors d'implanter l'interface ainsi :

```
void Move(float dx, float dy, float dz = 0.0);
```

pour que tous les codes existants utilisant l'ancienne interface, à deux arguments :

```
Cercle c1;
float dx, dy;
...
c1 .Move (dx, dy) ;
```

restent recompilables en l'état !

Surcharge des opérateurs

1. Introduction

Quiconque a déjà programmé, en Fortran, en Pascal, en C, connait la notion d'*opérateur*. Il s'agit d'une notation symbolique spécifiant une opération à faire, un traitement, dont la nature peut varier en fonction des arguments ou opérandes.

Ainsi, dans tous les langages, l'écriture **10 / 3** spécifie une division entière dont le résultat sera **3**, alors que **10.0 / 3** spécifie une division réelle, résultat **3.3333**. C'est le compilateur qui va, à partir d'un même symbole opératoire et selon la nature des opérandes, déterminer le traitement à effectuer.

C++ permet de définir des opérations spécifiques, utilisant les notations symboliques du langage, sur des opérandes a priori inconnus des compilateurs, des objets par exemple. Le terme consacré, *surcharge d'opérateur*, est sans doute un peu abusif. On devrait plutôt parler de *définition d'opérateur* puisque cela s'applique à des opérations inconnues.

Les domaines d'application sont essentiellement mathématiques, calculs en complexes, algèbre linéaire, etc.

2. Arithmétique complexe

On se propose de développer cette notion, en construisant une classe d'objets *nombres complexes*, lesquels n'existent pas en standard en C ou C++.

On construit des objets à deux attributs, parties réelle et imaginaire. On va choisir d'implanter trois constructeurs :

- un constructeur par défaut, classique,
- un constructeur avec initialisateurs,
- un constructeur dit de copie, permettant de créer un objet par clonage d'un autre, passé par référence.

Interface (fichier **complex . h**) :

```
class complex
{
// Constructeurs
public:
complex();
complex(float real, float imag);
complex(complex& model);
// Attributs private:
float pR, pI;
};
```

Implémentation (fichier **complex.cxx**) :

```
#include "complex.h"
/* Constructeur par defaut */
complex::complex()
{
pR = pI = 0.0;
}
/* Constructeur initialisateur */
complex::complex(float real, float imag)
{
pR = real;
pI = imag;
}
/* Constructeur cloneur */
complex::complex(complex& model)
{
pR = model.pR;
pI = model.pI;
}
```

On dispose donc d'une base qui va permettre d'instancier des objets de différentes manières :

```
complex c1;
complex c2(3.5, 0);     // Avec initialisation
complex c3(c2);         // Clonage sur c2
...
```

Que peut-on faire maintenant, de ces objets ? Pas grand-chose en fait ! On peut vouloir accéder aux attributs, privés, donc par un mécanisme d'accesseurs, **SetReal**, **SetImag**, **GetReal**, **GetImag**.

La technique manque un peu d'élégance et on va très vite se trouver face à du code comme :

```
//  Affectation  :  c1  =  c3
c1.SetReal(c3.GetReal());
c1.SetImag(c3.GetImag());
// Somme composite : c1 += c2
c1.SetReal(c1.GetReal()+c2.GetReal());
c1.SetImag(c1.GetImag()+c2.GetImag());
```

Tout ce qu'on peut dire de la monstruosité précédente est que cela fonctionne !

3. Opérateurs sur la classe

Heureusement, C++ permet de définir des opérateurs **=**, ou **+=**, valables pour des objets de la classe **complex**. On ajoutera dans l'interface :

```
class complex
{
...
// Operations
public:
void operator=(complex& arg);
void operator+=(complex& arg);
```

et, dans l'implémentation:

```
/* Affectation */
void complex::operator=(complex& arg)
{
pR=arg.pR;
pI=arg.pI;
}
/* Somme composite */
void complex::operator+=(complex& arg)
{
pR+=arg.pR;
pI+=arg.pI;
}
```

On dispose maintenant d'outils propres, permettant d'écrire du code qui ressemble à de l'arithmétique en C :

```
complex c1(1, 0);
complex c2(2.5, -1);
c2 += c1;
c1 = c2; ...
```

4. Associativité

En C++ comme en C, certains opérateurs sont associatifs, en particulier l'affectation, ce qui permet d'écrire des choses telles que :

```
int i, j, k;
i = j = k = 3;
```

Cela fonctionne parce que l'affectation est une opération qui retourne son opérande de droite. Syntaxiquement, le résultat d'une affectation est donc un opérande et peut figurer dans une expression.

Modifions donc notre opérateur d'affection pour qu'il retourne son opérande de droite, à savoir une référence sur un objet. L'interface devient :

```
class complex
{
     ...
     complex& operator=(complex& arg);
```

et l'implémentation :

```
complex& complex::operator=(complex& arg)
{
pR = arg.pR;
pI = arg.pI;
return arg;
}
```

Maintenant, des écritures telles que :

```
complex c1, c2;
complex c3(2.5, -1);
c1 = c2 = c3;
```

sont licites.

NB : dans l'écriture de l'opérateur **=**, on retourne l'opérande de droite. Si l'on a besoin, dans un opérateur, de retourner l'opérande de gauche, c'est à dire l'objet courant, on se souviendra du pointeur **this**.

Attention, **this** est un pointeur sur nous, mais ce n'est pas nous ! Nous, c'est l'objet pointé par **this**, et donc :

```
complex& complex::operator=(complex& arg)
{
pR = arg. pR;
pI = arg.pI;
return *this; // Mais oui !
}
```

5. Surcharge de sélection

Encore mieux, la surcharge permettant au compilateur de choisir telle ou telle méthode en fonction des arguments est applicable aux opérateurs.

Implantons donc la multiplication composite, ***=**, entre complexes ou entre complexe et scalaire.

On ajoute dans l'interface :

```
class complex
{ ...
void operator*=(float arg);
void operator*=(complex& arg);
```

et on code, dans l'implémentation :

```
void complex::operator*=(float arg)
{
pR *= arg; pI *= arg;
}
void complex::operator*=(complex& arg)
{
float R = pR * arg.pR - pI * arg.pI;
float I = pR * arg.pI + pI * arg.pR;
pR = R;
pI = I;
}
```

On peut maintenant écrire :

```
complex c1(1,1);
complex c2(5,0);
c2 *= c1;
c2 *= 0.5;
```

et le compilateur choisira l'opération ad-hoc en fonction des opérandes exactement comme dans le cas de l'arithmétique scalaire.

6. Objets temporaires

Dans les exemples précédents, on a sournoisement contourné un problème en implantant des opérateurs composés, **+=** ou ***=**, mais pas des opérateurs simples, **+** ou ***.**

Que se passe-t-il dans une expression arithmétique :

```
int i, j, k;
i = 3 + j + k;
```

En pratique, le compilateur utilise une ou plusieurs variables intermédiaires, anonymes (le plus souvent des registres du CPU), pour conserver les résultats partiels des opérations et les utiliser comme opérandes pour les opérations suivantes.

L'expression ci-dessus est traitée, en interne, comme :

```
int i, j, k;
int __1, __2;                    // Temporaires
__1 = 3 + j;
__2 = __1 + k; i = __2;
```

On utilisera le même principe en créant, dans les opérateurs, un objet temporaire résultat. Par contre, on ne devra plus implanter des opérateurs internes à la classe, utilisant l'objet courant, mais des opérateurs externes à deux opérandes. Pour qu'ils puissent tout de même accéder aux attributs, ils seront déclarés **friend**.

Ajoutons une addition dans l'interface :

```
class complex
{
...
friend complex operator+(complex& a, complex& b);
```

et dans l'implémentation :

```
complex operator+(complex& a, complex& b)
{
float R = a.pR + b.pR;
float I = a.pI + b.pI;
complex result(R, I); // Nouvel objet
return result;
}
```

NB : le type retour est bien un objet, **complex**, et non une référence, **complex&**. Le compilateur doit être prévenu que l'opération a créé une instance, temporaire, et qu'elle devra être détruite après utilisation, donc en fin de traitement de l'expression.

On dispose maintenant d'une véritable addition :

```
complex c1(1, 0);
complex c2(1, 1);
complex c3(c2);
complex c4;
c4 = c1 + c2 + c3;
```

7. Remarques

Les exemples précédents illustraient les principes de la surcharge, ou redéfinition d'opérateurs.

Tous les opérateurs de C peuvent être redéfinis, arithmétiques mais aussi booléens. On pourrait comparer des nombres complexes, par des écritures telles que :

```
complex c1, c2;
...
if( c1 == c2 ) ...
```

en implantant des opérateurs **==**, **~ ! =**, etc.

Il est conseillé de conserver un minimum de bon sens. Même si C++ le permet, ce n'est PAS une bonne idée d'implanter une addition de complexes sur l'opérateur ***** et une multiplication sur l'opérateur **+**, d'autant que les priorités d'évaluation restent celles de l'arithmétique :

(a + b * c)

s'évalue selon **(a + (b * c))**