Chapitre 6

La surcharge

Le mot **polymorphisme** veut dire qui a plusieurs formes. Le polymorphisme, sous une forme embryonnaire, se rencontre pratiquement dans tout langage de programmation. Par exemple l'opérateur de division / a deux formes en langage C version ANSI : si les deux opérateurs sont des entiers alors le résultat est un entier (le quotient exact dans la division euclidienne du premier entier par le second) ; c'est un réel si l'un des opérandes est un réel.

Il ya deux sortes de polymorphisme évolué pour les fonctions en C++ : le **polymorphisme paramétré**, grâce aux fonctions et aux classes génériques, et la **surcharge** (*overloading* en anglais). Dans le cas du polymorphisme paramétré, les types sont laissés non spécifiés mais le code est le même pour toutes les fonctions, il est donc défini une fois pour toutes. Dans le cas de la surcharge, le même symbole de fonction peut jouer plusieurs rôles mais on définit le code au cas par cas.

Nous allons nous intéresser à la surcharge dans ce chapitre. Nous commencerons par le cas général de la surcharge des fonctions. Nous verrons ensuite le cas des *opérateurs* qui sont des fonctions particulières. Ces deux premiers cas ne font pas référence à la programmation objet ; nous terminerons par la surcharge liée à la programmation orientée objet, en voyant en particulier le cas de plusieurs constructeurs et le problème de la surcharge lors de l'héritage.

6.1 Surcharge de fonction

Introduction.- À chaque fois que nous devons donner un nom à une entité, nous choisissons un identificateur. Nous avons insisté jusqu'à maintenant sur le fait que cet identificateur ne doit pas avoir été choisi pour deux entités différentes. Mais psychologiquement nous avons quelquefois envie de nommer de la même façon certaines entités. Ceci est le cas, par exemple, pour l'addition quel que soit l'ensemble de nombres (entier ou réel) ou pour la fonction d'échange ou de tri quel que soit le type des éléments...

Nous allons voir que ce qui n'est pas possible dans la plupart des langages de programmation, tel que le langage C, l'est en C++ grâce à la notion de *surcharge*.

Principe de la surcharge.- Si on donne le même nom à plusieurs fonctions, comment savoir la fonction qui sera choisie au moment de l'exécution ?

En fait une fonction n'est pas seulement caractérisée par son nom mais aussi par sa **signature**, c'est-à-dire par la liste des types de ses arguments. En C++ on peut donner le même nom à deux fonctions différentes à condition que les signatures soient différentes.

Remarque.- 1^o) En mathématiques intervient également dans la signature le type de retour (c'est-à-dire de l'ensemble d'arrivée). Il est important, par contre, que le type de retour ne fasse pas partie de la signature dans le cas d'un langage de programmation. En effet le choix de la fonction à évaluer se fonde sur l'analyse des types des paramètres effectifs.

- 2^o) Pour la même raison, il ne faut pas qu'il y ait ambiguïté sur la signature (n'oublions pas que 2, par exemple, est à la fois une constante `char`, `int` et `float`).

Exemple 1.- (**Même nombre d'arguments**)

Supposons que nous voulions définir une fonction binaire d'élévation à la puissance. Le code que nous choisirons n'est pas le même suivant que l'exposant est entier ou réel. En effet pour x réel et n entier naturel, d'une part, et pour x réel strictement positif et y réel, d'autre part, nous avons intérêt à utiliser les définitions respectives suivantes :

$$x^n = x \times x \times ... \times x \text{ avec } n \text{ occurrences de } x,$$
$$x^y = e^{y.lnx}.$$

Ceci nous conduit au programme suivant :

```
// exponent.cpp

#include <iostream.h>
#include <math.h>

float puiss(float x, int n)
   {
   float y;
   int i;
   cout << "\n puissance entiere : ";
   y = 1;
   for (i = 1; i <= n; i++) y = y*x;
   return y;
   }
```

```
float puiss(float x, float y)
   {
   cout << "\n puissance reelle : ";
   return exp(y*log(x));
   }

void main(void)
   {
   float x, y;
   int n;
   x = 2;
   n = 3;
   cout << puiss(x,n);
   y = 3;
   cout << puiss(x,y) << "\n";
   // cout << puiss(3.0, 3.1) << '\n';
   }
```

On remarquera que si l'on enlève la mise en commentaire de la dernière ligne, on obtient une erreur de compilation, indiquant que l'appel de la fonction surchargée est ambiguë, puisqu'il y a deux candidats possibles. Rappelons en effet que 3.1 n'est pas de type *float* mais de type *double*.

Cette ambiguïté disparaît si on remplace cette ligne par :

```
cout << puiss(3.0,(float) 3.1) << '\n';
```

Exemple 2.- (**Nombre d'arguments différents**)

La surcharge est souvent utilisée pour les constructeurs lors de la définition d'une classe. Considérons, par exemple, une classe *string* comprenant un tableau chaîne de caractères et sa longueur effective. Nous avons envie de permettre plusieurs constructeurs : un constructeur sans argument initialisant avec le mot vide et la longueur nulle un mot d'au plus 255 caractères, un constructeur avec un argument entier initialisant avec le mot vide et la longueur nulle un mot d'un nombre maximum de caractères spécifié par cet entier et un constructeur avec comme argument une chaîne de caractères (au sens du langage C) initialisant avec ce mot et la longueur de ce mot un mot de nombre de caractères maximum la longueur du mot entré.

On a, par exemple, le programme suivant :

```
// string.cpp

#include <iostream.h>
#include <string.h>

class string
   {
   char *s;
   int len;
   public:
   string(void);
   string(int);
   string(char *);
   ~string(void);
   void affiche(void);
```

```
   void concat(string, string);
   };

string::string(void)
   {
   s = new char[256];
   s[0] = '\0';
   len = 0;
   }

string::string(int n)
   {
   s = new char[n+1];
   s[0] = '\0';
   len = n;
   }

string::string(char *p)
   {
   len = strlen(p);
   s = new char[len + 1];
   strcpy(s, p);
   }

string::~string(void)
   {
   delete [] s;
   len = 0;
   }

void string::affiche(void)
   {
   cout << s << '\n';
   }

void string::concat(string a, string b)
   {
   len = a.len + b.len;
   strcpy(s, a.s);
   strcat(s, b.s);
   }

void main(void)
   {
   string a, b(5), c("essai");
   a.affiche();
   b.affiche();
   c.affiche();
   a.concat(b, c);
   a.affiche();
```

```
}
```

Rappelons la façon d'utiliser le constructeur sans argument. On a :

```
string a;
```

sans parenthèses et non :

```
string a();
```

6.2 Surcharge d'opérateur

Un principe de la programmation orientée objet est qu'un type défini (autrement dit une classe) a les mêmes droits qu'un **type natif**, c'est-à-dire un type prédéfini tel que celui des entiers. Un tel type a donc le droit d'utiliser des **opérateurs**, c'est-à-dire des méthodes binaires pour lesquelles on utilise la notation infixe :

```
c = a + b,
```

et non la notation préfixe :

```
c = +(a,b).
```

On doit donc pouvoir surcharger les opérateurs pour que ceux-ci puissent être des méthodes.

6.2.1 Un exemple

Problème.- On veut définir une classe pour les nombres complexes et noter de façon habituelle l'addition et la multiplication.

Un programme.- Ceci conduit au programme suivant.

```
// complex.cpp

#include <iostream.h>

class complex
    {
    float x, y;
    public:
    void init(float, float);
    void affiche(void);
    complex operator + (complex);
    complex operator * (complex);
    };

void complex::init(float re, float im)
   {
   x = re;
   y = im;
   }

void complex::affiche(void)
```

```
   {
   cout << x <<" + " << y << ".i";
   }

complex complex::operator + (complex b)
   {
   complex c;
   c.x = x + b.x;
   c.y = y + b.y;
   return c;
   }

complex complex::operator * (complex b)
   {
   complex c;
   c.x = x*b.x - y*b.y;
   c.y = x*b.y + y*b.x;
   return c;
   }

void main(void)
   {
   complex a, b, c, d;
   a.init(2, 3);
   b.init(1, 4);
   c = a + b;
   d = a*b;
   a.affiche();
   cout << " + ";
   b.affiche();
   cout << " = ";
   c.affiche();
   cout << "\n(";
   a.affiche();
   cout << ").(";
   b.affiche();
   cout << ") = ";
   d.affiche();
   cout << "\n";
   }
```

6.2.2 Mise en place

Syntaxe.- La syntaxe est la suivante :

`type` **operator** `symbole (arguments);`

où `type` est le type de retour, **operator** un mot clé du langage C++, `symbole` un des symboles d'opérateurs possibles et arguments les arguments habituels d'une fonction (type et variable).

Tout symbole possède une **arité**, à savoir un entier naturel (dans les faits seulement un ou deux). Le nombre d'arguments doit être conforme à cette arité.

Les symboles d'opérateurs du C++.- On peut redéfinir tous les opérateurs existants du C++ (il n'est pas possible de créer de nouveaux symboles), à condition qu'ils portent sur un objet au moins. Les symboles d'opérateurs du C++ sont :

les opérateurs binaires

- d'addition + ;
- de soustraction - ;
- de multiplication * ;
- de division / ;
- de modulo % ;

et les opérateurs unaires

- d'affectation = ;
- d'indexation [ ] ;
- d'allocation de mémoire `new` ;
- de désallocation de mémoire `delete`.

Remarque.- Bien que cela ne présente pas d'intérêt, on peut écrire fonctionnellement :

c = operator+(a,b) ;

au lieu de :

c = a + b ;

6.3 Surchage et héritage

6.3.1 Surcharge des fonctions membre

Introduction.- Nous avons vu la notion d'héritage, avec l'ajout de nouveaux membres pour une classe dérivée par rapport à la classe de base, que ce soit des membres attributs ou des fonctions membre.

Une autre façon de concevoir une classe dérivée consiste à redéfinir l'action d'une fonction membre.

Exemple.- Nous pouvons définir les classes `point` et `pointcol` (pour point coloré) ayant une fonction `affiche()` de même signature :

```
// sur_mem.cpp

#include <iostream.h>

class point
    {
    protected:
    float x, y;
    public:
    void init(float, float);
```

```
    void affiche(void);
    };

class pointcol: public point
    {
    short n;
    public:
    void init(float, float, short);
    void affiche(void);
    };

void point::init(float abs, float ord)
    {
    x = abs;
    y = ord;
    }

void point::affiche(void)
    {
    cout << '(' << x << ',' << y << ')';
    }

void pointcol::init(float abs, float ord, short col)
    {
    point::init(abs, ord);
    n = col;
    }

void pointcol::affiche(void)
    {
    point::affiche();
    cout << " de couleur " << n;
    }

void main(void)
    {
    class pointcol p;
    p.init(2.0, 3.0, 4);
    cout << "\n";
    p.affiche();
    cout << "\n";
    }
```

La fonction `affiche()` prise en compte est bien celle de la classe `pointcol`.

6.3.2 Problème de la liaison statique

Introduction.- Considérons le programme principal suivant, avec la définition des classes ci-dessus :

```
// static.cpp
#include <iostream.h>

class point;
class pointcol;

void main(void)
   {
   point P;
   pointcol PC;
   point *pp;
   pp = &P;
   pc.init(2.0, 3.0, 4);
   pc.affiche();
   cout << "\n";
   pp = &PC;
   pp->affiche();
   cout << "\n";
   }
```

L'exécution donne :

```
   (2,3) de couleur 4
   (2,3)
```

autrement dit l'affichage du pointeur `pp` se fait comme celui d'un point non coloré et non comme celui d'un point coloré, contrairement à ce que l'on pourrait attendre.

Explication.- On dit que la **liaison** est **statique** par défaut en C++. Comme le pointeur est d'abord déclaré pour un point non coloré, c'est la fonction `affiche()` correspondante qu'il prendra toujours en compte, même si on lui donne l'adresse d'un élément d'une sous-classe.

Conséquences.- Le fait que le typage soit statique (et non **dynamique**) a des conséquences, en particulier lorsqu'on passe un argument d'une fonction par adresse.

6.3.3 Fonction virtuelle et liaison dynamique

Introduction.- Pour que la liaison soit dynamique il faut, en C++, déclarer les fonctions concernées en tant que **fonctions virtuelles**.

Syntaxe.- On fait précéder la déclaration (ou la définition, dans le cas d'une fonction en ligne) de la fonction par le mot clé `virtual`.

Exemple.- Réécrivons le programme précédent en utilisant une fonction virtuelle :

```
// dynamic.cpp
#include <iostream.h>
```

```
class point
   {
   protected:
   float x, y;
   public:
   void init(float, float);
   virtual void affiche(void);
   };

class pointcol: public point
   {
   short n;
   public:
   void init(float, float, short);
   void affiche(void);
   };

void point::affiche(void);

void main(void)
   {
   point P;
   pointcol PC;
   point *pp;
   pp = &P;
   PC.init(2.0, 3.0, 4);
   PC.affiche();
   cout << "\n";
   pp = &PC;
   pp->affiche();
   cout << "\n";
   }
```

L'exécution donne :

```
   (2,3) de couleur 4
   (2,3) de couleur 4
```

autrement dit l'affichage du pointeur `pp` se fait bien, cette fois-ci, comme celui d'un point coloré.

Exercice.- *L'exemple classique est celui de l'aire d'une figure. On considère une classe de base* `figure` *et des classes correspondant à des types de figures (carré, rectangle, cercle...). Définir une fonction virtuelle aire qui sera redéfinie pour chaque classe héritée.*

6.4 Constructeur de copie et surcharge de l'affectation

Introduction.- L'implémentation des classes d'objets dynamiques conduit souvent à une erreur d'exécution assez fine. Il faudra définir un constructeur spécial, appelé **constructeur de copie**, et surcharger l'affectation pour l'éviter.

Exemple.- Nous voulons implémenter une classe d'ensembles (finis) d'entiers naturels. Une façon de faire est de considérer qu'un tel ensemble est un tableau `tab` de booléens (donc de *int* en C++) : un entier i appartient à l'ensemble si `tab[i]` est vrai, c'est-à-dire est un entier non nul. Bien entendu, comme d'habitude pour les tableaux, nous aurons une donnée `max`, de type *int*, qui représentera le (successeur du) plus grand entier pouvant apparaître dans cet ensemble.

Les méthodes que nous voulons implémenter sont un constructeur (ayant un paramètre entier donnant la valeur de `max`, l'ensemble étant initialisé à l'ensemble vide), un destructeur, une fonction d'ajout d'un élément, une fonction de retrait, une fonction de test pour savoir si un entier appartient à un ensemble, une fonction donnant le nombre d'éléments de l'ensemble, une fonction d'affichage et les opérations classiques sur les ensembles (union, intersection et différence).

Comme d'habitude, implémentons ces méthodes en les testant une par une. Au début tout va bien. Nous avons, par exemple, le programme suivant :

```
#include <iostream.h>

class ensemble
   {
   //attributs
   int max;
   int *tab;
   //methodes
   public:
   ensemble(int n);
   ~ensemble();
   void ajout(int el);
   int card();
   void affiche();
   };

ensemble::ensemble(int n)
   {
   max = n;
   tab = new int[max];
   int i;
   for (i = 0; i < max; i++)
      tab[i] = 0;
   }

ensemble::~ensemble()
   {
   max = 0;
   delete [] tab;
   }
```

```
void ensemble::ajout(int el)
   {
   if (el < max) tab[el] = 1;
   }

int ensemble::card()
   {
   int i, n;
   n = 0;
   for (i = 0; i < max; i++)
      if (tab[i]) n++;
   return n;
   }

void ensemble::affiche()
   {
   cout << '{';
   int i;
   for (i=0; i < max; i++)
      if (tab[i]) cout << i << ", ";
   if (card() == 0) cout << " }'';
   else  cout << "\b\b}";
   }

void main()
   {
   ensemble A(16);
   cout << "Au debut A = ";
   A.affiche();
   A.ajout(2);
   A.ajout(7);
   A.ajout(15);
   A.ajout(0);
   cout << "\nApres A = ";
   A.affiche();
   cout << "\ncard(A) = " << A.card() << '\n';
   }
```

L'exécution est correcte. On obtient :

```
Au début A = { }
Après A = {0, 2, 7, 15}
card(A) = 4
```

Par contre, un problème d'exécution apparaît lorsque nous implémentons l'opération d'intersection. Nous avons, par exemple, le programme suivant :

```
// set2.cpp
#include <iostream.h>
```

```
class ensemble
    {
    //attributs
    int max;
    int *tab;
    //methodes
    public:
    ensemble(int n);
    ~ensemble();
    void ajout(int el);
    int card();
    void affiche();
    ensemble inter(ensemble B);
    };

ensemble ensemble::inter(ensemble B)
   {
   int n;
   if (max < B.max) n = max;
   else n = B.max;
   ensemble C(n);
   int i;
   for (i=0; i < n; i++)
      if ((tab[i]) && (B.tab[i])) C.tab[i] = 1;
   return C;
   }

void main()
   {
   ensemble A(16);
   ensemble B(25);
   ensemble C(30);

   A.ajout(2);
   A.ajout(7);
   A.ajout(15);
   A.ajout(0);
   cout << "A = ";
   A.affiche();

   B.ajout(6);
   B.ajout(7);
   B.ajout(0);
   B.ajout(23);
   cout << "\nB = ";
   B.affiche();

   C = A.inter(B);
   cout << "\nC = ";
   C.affiche();
```

```
cout << "\n";
}
```

La compilation et le chargement se passent sans indiquer d'erreur. L'exécution sur Borland C++ 4.51 se passe sans problème. Avec Visual C++ 4.0 on obtient des messages incompréhensibles. Avec g++ on commence par les trois lignes suivantes :

```
A = {0, 2, 7, 15}
B = {0, 6, 7}
C = {7}
```

et le programme ne s'arrête pas. Remarquons que, même dans ce dernier cas, le résultat de l'intersection n'est pas correct.

Explication.- On se sert d'un ensemble local `C` dans l'implémentation de la fonction `inter`. Cet ensemble est calculé correctement (faites-le afficher à ce moment !) puis on le transmet à l'extérieur grâce à `return`. Ceci a pour effet de copier les données, c'est-à-dir `max` et `tab` ; c'est bien ce que l'on veut.

Lorsqu'on quitte cette fonction, cependant, nous sommes à la fin d'un bloc. Il est donc fait appel automatiquement à la fonction de destruction. On désalloue alors la partie de la mémoire affectée à `C.tab`, ce qui est bien ce que l'on veut. Mais ceci a aussi pour conséquence de désallouer la mémoire affectée à la valeur de retour, puisque c'est la même. On obtient ainsi ce que l'on appelle une **référence folle**.

Remède.- Pour remédier à cette situation, toute classe `X` d'**objets dynamiques** (c'est-à-dire ayant au moins un attribut de type pointeur) devrait comporter au moins :

- un **constructeur par défaut** de la forme `X::X()` ;
- un **constructeur de copie** de la forme `X::X(const X &)` ;
- une surcharge de l'opérateur d'affectation de la forme :

$$\texttt{X} :: \texttt{operator} = (\texttt{const X}\&);$$

- un destructeur `~X()`.

Nous n'avons rien à ajouter en ce qui concerne le constructeur par défaut ou le destructeur. Le constructeur de copie et la nouvelle affectation ont pour rôle de copier les données correctement.

Cas de notre exemple.- Pour notre classe `ensemble` nous rajoutons donc trois méthodes (le destructeur existant déjà) :

```
// set3.cpp
#include <iostream.h>

class ensemble
   {
   //attributs
   int max;
   int *tab;
   //methodes
   public:
   ensemble();
   ensemble(int n);
```

```
   ensemble (const ensemble&);
   ~ensemble();
   void ajout(int el);
   int card();
   void affiche();
   ensemble& ensemble::operator=(const ensemble& E);
   ensemble inter(ensemble B);
   };

ensemble::ensemble()
   {
   max = 0;
   tab = new int[1];
   tab[0] = 0;
   }

ensemble::ensemble (const ensemble& E)
   {
   max = E.max;
   tab = new int[max];
   int i;
   for (i=0; i < max; i++)
      tab[i] = E.tab[i];
   }

ensemble& ensemble::operator=(const ensemble& E)
   {
   if (this == &E) return *this;
   delete [] tab;
   max = E.max;
   tab = new int[max];
   int i;
   for (i=0; i < max; i++)
      tab[i] = E.tab[i];
   return *this;
   }
```

Les reste est inchangé. Cette fois-ci l'exécution se passe correctement.

Commentaires.- Le constructeur par défaut est réduit au minimum. Le constructeur de copie ne transmet pas la valeur de `E.tab` : le nouveau pointeur `tab` est initialisé (donc occupe un autre emplacement mémoire), la même quantité de mémoire est retenue puis on effectue une copie effective du contenu du tableau `tab`.

La définition de l'opérateur d'affectation est semblable à celle du constructeur par défaut à deux points près : d'une part, il faut faire attention au cas (improbable) où la valeur gauche est égale à la valeur droite (l'affectation `E = E`, dans ce cas il suffit de ne rien faire) ; d'autre part, on désalloue la mémoire occupée précédemment.

Le pointeur this.- On remarquera un petit problème au moment de renvoyer la valeur : comment nommer celle-ci ? En C++ on utilise un pointeur sur l'objet dont on parle, appelé `this`.

Index