partie 3 : Mécanisme de surcharge

MOOC Langage C++ INSA

Source : Ce document est en partie extrait du livre : du langage C au C++ par Thierry Monteil, Vincent Nicomette, François Pompignac, Saturnino Hernando, Presses Universitaires du Midi ISBN 978-2-8107-0054-7

1 Surcharge

On parle de surcharge d'un nom de fonction quand le même nom de fonction peut être utilisé pour désigner plusieurs fonctions. La différence entre les fonctions provient :

— du nombre de paramètres différents d'un appel à l'autre,

— de deux (ou plusieurs) types qui sont acceptés pour le i^{ieme} paramètre.

1.1 Principe

Le choix du code à exécuter se fait en fonction des paramètres. En C, l'opérateur "/" est surchargé. Il représente à la fois la division entière et la division réelle. En C++, le programmeur peut surcharger ses propres fonctions ainsi que les opérateurs définis dans le langage.

La surcharge est un mécanisme permettant le polymorphisme. Le choix du bon code est réalisé à la compilation. Il s'agit d'un mécanisme purement syntaxique. Nous verrons des cas particuliers où le choix doit être retardé jusqu'à l'exécution : notion de méthode virtuelle(cf partie **?? ??**).

Il existe deux types de surcharge :

— Surcharge avec valeurs des paramètres par défaut. L'affectation des paramètres par défaut se fait sans interruption, **Exemple:**

```
/* string est une classe représentant les chaînes de caractères en C++, cf chapitre ??*/
string g(char a1 = 0, char a2 = 0, string a3 = string("")) {
     return a1 + a2 + a3 ;}
...
     g(); // renvoie le string "" (null)
     g('a');// renvoie le string "a"
     g('a','b');/* renvoie le string "Ã" caractère de code ascii égal à la somme des codes ascii
de 'a' et 'b' */
     g('a','b',"ccc"); renvoie le string "Ãccc"
     /* g('a',''ccc''); appel illégal car le deuxième paramètre de type caractère a été omis */
```

— Surcharge par redéfinition explicite, on doit assurer qu'il n'y aura pas d'ambiguïté à l'appel. Une fonction est définie par sa signature (son nom + le type de ses paramètres, indépendamment du type de résultat). Une ambiguïté est une situation de surcharge pour laquelle un appel peut être traité par deux (ou plus) définitions, **Exemple:**
Les définition suivantes sont ambiguës pour l'appel f(1)

```
float f(char a) {return 1.1;}
float f (float a) {return 1.2;}
```

En effet, cet appel peut être exécuté par les deux définitions car les entiers peuvent être convertis en “char” ou en “float” sans aucune préférence.

Exemple:

Cet exemple est correct.

```
float f(char a) {return 1;}
float f (String a) {return 1.1;}
```

L’appel f(1) utilise ici la définition f(char a).

Il peut y avoir ambiguïté entre les deux types de surcharge dans l’exemple ci-dessous. **Exemple:**

```
float g(char a1 = 0, char a2 = 0, string a3 = String("")) {
     return a1 + a2 + a3 ;}
float g(char a1, char a2) {
     return a1 + a2 ;}
```

Les deux définitions entraînent une ambiguïté pour l’appel “g(’a’,’b’) ;”. Cette ambiguïté ne sera levée par le compilateur que si un appel à la fonction g écrit dans le programme est ambigu, ce qui fait que l’on peut avoir un programme qui fonctionne correctement jusqu’au jour où l’on modifie un appel à la fonction g qui crée une ambiguïté.

1.2 Surcharge des opérateurs

Le C++ permet de surcharger aussi les opérateurs standards du langage. Dans ce cas, les règles de priorité et d'association sont conservées. La syntaxe est aussi conservée, en particulier le nombre d'arguments sera le même. Ceci offre la possibilité de manipuler de la même façon toutes les variables indépendamment de leur type (simple ou structuré). On distingue :

- les opérateurs d'affectation : '=' (le seul défini par défaut : copie champ/champ) (et '+=', '-=','*='...),
- les opérateurs unaires (inc/dec) : '++', '--',
- les opérateurs de comparaison : '==' (et '!='),
- les opérateurs d'entrée et de sortie (streams) : '<<' et '>>',
- les opérateurs arithmétiques : '+', '/', etc,
- les opérateurs d'appel de fonction "()", d'indice "[]", d'accès "->". Pour ces trois derniers, ils doivent obligatoirement être définis dans la classe,
- les opérateurs d'allocation et de désallocation : new et delete.

Ces opérateurs sont de deux familles :

- Ceux qui ont besoin d'accéder aux données privées et qui seront plutôt définis à l'intérieur de la classe (ou déclarés amis, voir dans la partie **?? ??**),
- Ceux qui n'accèdent pas aux données privées et qui peuvent donc être définis à l'extérieur de la classe.

Beaucoup d'opérations renverront un objet afin d'être associatives. Par exemple a=b=c s'interprète ainsi : la variable a est affectée avec le résultat de b=c. La définition d'un opérateur se fera avec le mot-clé operator.

1.2.1 Opérateur d'affectation

Par défaut, pour une classe, l'affectation se fait membre à membre. **Exemple:**

```
#include <iostream>
#include <math.h>
using namespace std;
class complexe {
     public:
          float reel() { return r;}
          float imag() { return i;}
          //on ne peut pas confondre les 2 constructeurs
          complexe() {vide = true;}
          complexe(float x,float y) {
               vide = false; //Pb si x et y non initialisées !
               r=x; i=y;
          }
          complexe(complexe & autre) { // & obligatoire
               vide = autre.vide; r=autre.r; i=autre.i;}
          float norme() {
               //...inchangé
          }
          //set est remplacée par :
          //1. le constructeur paramètre; 2. l'operateur standard : =
```

```
            complexe & operator = (const complexe & autre) {
                  r = autre.r; i = autre.i;
                  vide = autre.vide; //possible bien que vide est privé.
                  return (*this);
            }
      private : // protège les variables vide, r et i
            bool vide;
            float r,i;
};
int main (){
      complexe y;
      y= complexe(0,1); /* équivalent du y.set(0,1); construction d'un
      complexe non nommé utilisé pour modifier y*/
      cout << y.norme() << endl;
      return 0;
}
```

Comme pour tous les opérateurs définis dans une classe, on peut écrire :
`a=b ou a.operator = (b);`

Le format classique de cet opérateur sera :

nomclasse & operator = (const nomclasse & variable) {
  ...
  return (*this) ;
}

1.2.2 Opérateur d'incrémentation

Voici deux exemples de surcharge des opérateurs "+=" et "++" :

Exemple:

```
#include <iostream>
#include <math.h>
using namespace std;
class complexe {
     public:
     ...
          complexe & operator += (const complexe & autre) {
               r += autre.r; i += autre.i;
               //vide = autre.vide;
               return (*this);
          }
          complexe & operator ++() {
               ++r; ++i;
               return (*this);
          }
     private :
          bool vide;
          float r,i;
};
int main (){
     complexe y(0,1);
     y++;
     y += y += y;
     return 0;
}
```

Le C++ garde la notion de post ou préfixe. La différence se fait à la déclaration de l'opérateur. Le choix qui a été fait est le suivant :

- "nomclasse & operator ++() " sera la notation préfixée,
- "nomclasse & operator ++(int) " sera la notation postfixée, on peut remplacer int par n'importe quel type,
- des définitions équivalentes existent si on déclare ces opérateurs à l'extérieur de la classe.

1.2.3 Opérateur de comparaison

La comparaison se fait par surcharge de l'opérateur "==". La nouvelle définition de l'opérateur n'a pas besoin d'accéder aux variables privées en modification et peut donc tout à fait se faire à l'extérieur de la classe. **Exemple:**

```
#include <iostream>
#include <math.h>
using namespace std;
class complexe {
   //...inchangée
};

   bool operator == (const complexe& un, const complexe& autre) {
     return (un.reel()== autre.reel()) && (un.imag() == autre.imag());}

   int main (){
     complexe x,y;
```

```
      y= complexe(0,1);
      cout << y.norme() << endl;
      if (y == (x=y)) {cout << "OK\n";}
      else cout << "NOK\n";; //OK
      return 0;
}
```

1.2.4 Opérateur d'addition

Écrire $A = B + C$ revient à mettre le résultat de $B + C$ dans une variable temporaire qui est ensuite utilisée pour faire l'affectation avec A. Il faut donc être vigilant sur le retour de l'opérateur + qui ne doit pas renvoyer une variable périmée. Dans l'exemple proposé, on copie la variable *tamp* dans la pile et c'est la copie de *tamp* qui est ensuite utilisée pour l'affectation avec A. On utilisera aussi l'opérateur "+=" défini précédemment. Retourner une référence ou un pointeur sur *tamp* aurait été faux car ces données seront périmées. L'opérateur + peut se définir à l'extérieur de la classe car il ne modifie pas les opérandes intervenant dans l'addition. **Exemple:**

```
...
complexe operator + ( const complexe& un, const complexe& autre) {
      complexe tamp(un);
      return( tamp+=autre);
}
```

1.2.5 Opérateur d'indiçage

Il s'agit de l'opérateur "Operator[]". Cet opérateur est forcément membre d'une classe. Il est toujours défini avec un seul argument (type quelconque). **Exemple:**

```
class complexe {
      public:
            //...constructeurs
            //...operateur []
            float operator[] (int indice) {
                  if (indice == 1 ) return r;
                  return i; // retourne i pour tout entier != 1
            }
};
//Utilisation
int main () {
      complexe x (111,222) ;
      cout << x[1] ; // affiche 111
      cout << x[2] ; // affiche 222
      return 0;
}
```

Remarque:
Si on veut pouvoir réaliser une affectation sur la variable résultat de l'opérateur d'indiçage (appel à gauche d'une expression d'affectation), il faut retourner une référence.

```
...
            float & operator[] (int indice) {
                  if (indice == 1 ) return r;
                  return i; // retourne i pour tout entier != 1
            }
...
```

1.2.6 Opérateur d'appel de fonction

Cet opérateur peut être surchargé plusieurs fois suivant les types et le nombre de paramètres. Il sert parfois pour définir des itérateurs sur une classe. **Exemple:**

```
class Tableau_de_complexe {
     private:
          // tableau de complexes
          complexe tableau[10];

        public:
          //...constructeurs
          //...operateur ()
          float operator() (int indice1, int indice2) {
               //composante indice2 du complexe indice1
               //indice1 : de 1..n
               return tableau[indice1-1][indice2]; //
          }
};
//Utilisation
int main () {
     Tableau_de_complexe t(10); //dimension 10
     //saisie du tableau, non décrite
     //
     cout << t(1,2) ;
     // affiche la partie imaginaire
     // du "premier" complexe du tableau
     return 0;
}
```

1.2.7 Opérateur d'accès

"x-¿" correspond à "x.operator-¿". Il doit donc retourner quelque chose sur lequel l'opérateur "-¿" s'applique (un pointeur par exemple). La surcharge peut permettre d'obtenir des pointeurs intelligents qui font plus que retourner l'objet pointé.

Exemple:

```
...
class ptr_complexe{
     public:
          ptr_complexe() {ptr=NULL;}
          ptr_complexe (complexe * ref){ptr=ref;}
          complexe* operator->(){
               if (ptr == NULL){
                    cout << "erreur pointeur null\n";
                    exit(-1);
               }
               return (ptr);
          }
     private:
          complexe *ptr;
};

   int main (){
     complexe y;
```

```
      ptr_complexe pe, py (&y);
      y= complexe(2,1); // équivalent du y.set(0,1);
      cout << " norme py : " << py->norme()<< endl;
      cout << " norme pe : " << pe->norme() << endl;
      // affiche: norme py : 2.23607
      //erreur pointeur null
      return 0;
}
```

1.2.8 Opérateur de conversion

On peut définir l'opérateur qui permettra de réaliser des conversions explicites entre des types. Ceci revient à créer les opérateurs pour réaliser des "cast". C++ peut alors faire des conversions implicitement sur les classes de l'utilisateur. **Exemple:**
La conversion sera possible de X vers Y.

```
class Y{
public:
      int i;
      Y(int j) : i(j) {}; // conversion d'un int en Y
};
Y operator +(const Y &un, const Y &deux){
      return Y(un.i+deux.i);
}
class X {
public:
      int i ;
      X(int j) : i(j) {};
// constructeur
      operator Y() {return Y(i);} // convertit X en Y
};

   int main(){
     X x(1);
     Y y1(2),y2(2);
     /* x est converti automatiquement en Y puis l'addition entre Y est appelée */
     y1=x+y2;
     return(0);
}
```

Remarque:
Il est possible d'enchaîner au plus trois cast implicites. Cette chaîne de "cast" sera constituée avec un "cast" défini par l'utilisateur et deux "cast" prédéfinis en C++. Si l'on souhaite bloquer la conversion implicite pour une classe, il est nécessaire d'ajouter devant ses constructeurs le mot clé `explicit`.

1.2.9 Opérateur "void"

La définition d'un opérateur **void*()** dans une classe X, permet d'utiliser ses instances comme valeur de test. Ceci peut par exemple être utilisé dans une boucle "while" : **Exemple:**

```
#include <iostream>
using namespace std;
class X {
      public:
```

```
        int a ;
        X(int p=0) : a(p) {}
        operator void*() {return ((void*)(a != 10));}
};
int main () {
X x(0);
while (x) { cout << x.a << " "; x.a++;}
//affiche 0 1 2 3 4 5 6 7 8 9
return 0;
}
```

Cet opérateur est défini par exemple dans la classe “ios” (cette classe sera vue plus en détail dans le chapitre sur les flots, chapitre **??**) et permet d'écrire des boucles telles que : **Exemple:**

```
complexe x;
while (cin >> x ) { cout << "x = " << x;}
```

Cette boucle s'arrête dès que le format de lecture des complexes n'est plus respecté.