

Surcharge des fonctions et opérateurs

Introduction au C++ et à la programmation objet

E. Courcelle

CALMIP, UMS 3669

Décembre 2019

- 1 Surcharger les fonctions
- 2 Valeurs par défaut des arguments
- 3 Surcharger les opérateurs
- 4 Conversions

Surcharger une fonction ou une méthode

En C++, il est possible de déclarer et définir **plusieurs fonctions ayant le même nom**, à condition que les listes de leurs arguments diffèrent (types différents):

```
float flineaire (float x) {  
    float y = 3 * x + 2;  
    return y;  
}  
  
complexe flineaire (const complexe & x) {  
    complexe y(0,0);  
    y.set_r (3*x.get_r() + 2);  
    y.set_i (3*x.get_i());  
    return y;  
}
```

Les règles de surcharge

- Le type de la valeur de retour ne fait pas partie du prototype: Ainsi, il n'est pas possible de surcharger une fonction par une autre fonction qui aurait même nom et même liste d'arguments mais une valeur de retour différente.
- Dans le cas de méthodes d'objets, `const` fait partie du prototype, de sorte qu'on peut avoir une méthode `float f()` et une méthode `float f() const`
- La norme définit les règles permettant à partir d'une liste de paramètres de choisir l'une ou l'autre des fonctions.

Surcharger le constructeur

Le constructeur étant une fonction-membre parmi d'autres, il est possible de le surcharger.

Dans le cas du type complexe, cela permettra de remplacer la fonction-membre copie par le **constructeur de copie**.

Dès lors il est possible d'écrire `complexe a = b;`

```
class complexe {  
public:  
    complexe(float x, float y) : r(x), i(y) {};  
    complexe(const complexe& c) : r(c.r), i(c.i) {};  
private:  
    float r;  
    float i;  
    ...  
}  
  
main() {  
    const complexe j(0,1);  
    complexe A=j;  
}
```

Cas des fonctions ou des méthodes

On peut donner aux arguments des valeurs par défaut.

Une utilisation possible de cela est de réutiliser du code ancien, en étendant une fonction sans remettre en cause l'existant.

Par exemple avec la fonction `flineaire`:

```
float flineaire (float x, int a=3, int b=2) {  
    float y = a * x + b;  
    return y;  
}  
  
float c1 = flineaire(2.0);           // c1 vaut 8.0  
float c2 = flineaire(2.0,4);        // c2 vaut 10.0  
float c3 = flineaire(2.0,4,4);      // c3 vaut 12.0
```

ATTENTION

- Il n'y a pas d'arguments nommés en C++
- En conséquence, les arguments ayant une valeur par défaut sont **systématiquement** en fin de liste

Cas des constructeurs

On peut donner aux arguments des constructeurs des valeurs par défaut:

```
class complexe {
private:
    ...
public:
    complexe(float x=0, float y=0) {r=x;    i=y; _calc_module();};
    ...
};

main() {
    complexe c;           // sous-entendu initialiser a 0
    complexe c1(2);       // sous-entendu initialiser a (2,0) [reel]
    complexe c2 = {2};    // meme chose que ci-dessus
    complexe c3 = {2,3};
}
```

Conversions de types

On peut écrire et c'est une **facilité d'écriture**:

```
complexe C1 = 2;
```

Ce qui correspond à une **conversion** entre types (ici conversion depuis le type float vers le type complexe). Cette conversion peut aussi bien être nuisible si elle n'a pas de sens, elle peut dans ce cas être inhibée grâce au mot-clé **explicit**. Cela sera utile par exemple dans la définition de l'objet tableau:

```
class tableau {  
    ...  
public:  
    explicit tableau(int);  
};  
  
main() {  
    tableau T = 1000;    // Ne compile pas !  
    tableau T(1000);    // OK, cree un tableau de dimension 1000  
}
```

ATTENTION, Piège !

Si on ne met pas **explicit** dans le code ci-dessus, on risque d'avoir des conversions non souhaitées, qui conduisent à des erreurs à l'exécution. Ces erreurs seront détectées à la compilation si on les interdit par l'utilisation de **explicit**

Opérateurs et fonctions

Un opérateur est un appel de fonction écrit de manière plus lisible:

```
c = a + b :
```

pourrait s'écrire:

```
c = add(a, b) ;
```

mais ce ne serait pas très agréable à lire, surtout dans le cas de formules compliquées:

```
d = a + 2*(b+c) ; // Ecrit sous forme de formule  
d = add(a, mul(2, add(b, c))) ; // Moins agreable a lire
```


Opérateurs et fonctions (2)

- Surcharger un opérateur revient à surcharger une fonction.
- Cela permettra d'écrire des formules en utilisant des objets
- Par exemple $c = a + b$ où a, b, c sont des complexes

ATTENTION !!! Pièges !

- La surcharge des opérateurs ne **change pas les règles de priorité ou d'associativité**
- A chaque opérateur sa fonction avec son **nom réservé**
- Vous pouvez surcharger les opérateurs existants, **pas** en inventer de nouveaux !
- On peut mettre n'importe quoi dans le code de la fonction surchargée.
C'est au programmeur de **surcharger ses opérateurs intelligemment !**

Les quatre opérations

Opérateurs unaires

```
+= operator+=  
-= operator-=  
*= operator*=  
/= operator/=   
%= operator%=
```

Opérateurs binaires

```
+ operator+  
- operator-  
* operator*  
/ operator/  
% operator%
```

L'addition de complexes

Voici une première manière de définir les opérateurs `+` et `+=` sur les complexes.
La fonction `operator+` fait appel à une fonction amie, pour lui permettre d'accéder aux membres privés.

```
class complexe {
private:
    ...
public:
    ...
    complexe& operator+=(const complexe&);
    friend complexe operator+(const complexe& a, const complexe& b);
};

complexe& complexe::operator+=(const complexe& c) {
    r += c.r;
    i += c.i;
    return *this;
};

complexe operator+(const complexe& a, const complexe& b) {
    complexe c;
    c.r = a.r + b.r;
    c.i = a.i + b.i;
    return c;
}
```

L'addition de complexes (2)

Voici une seconde manière de définir les opérateurs `+` et `+=` sur les complexes. Elle est préférable à la précédente, car il y a moins de dépendences (et la fonction n'est plus une amie):

```
class complexe {
private:
    ...
public:
    ...
    complexe& operator+=(const complexe&);
};

complexe& complexe::operator+=(const complexe& c) {
    r += c.r;
    i += c.i;
    return *this;
};

complexe operator+(const complexe& a, const complexe& b) {
    complexe c=a;
    c += b;
    return c;
}
```

Incrémentation, décrémentation

Ces opérateurs (- -) et (+ +) sont utiles pour définir des **itérateurs**, permettant d'itérer sur des objets de type conteneur.

Il y a cependant une subtilité importante, utile pour reproduire exactement le comportement des opérateurs natifs du C correspondants:

Postindexation:

```
int i = 1;
j = i++;           // Incrémente et renvoie le vieux i
cout << i << j;   // Renvoie 2 1
```

Préindexation:

```
int i = 1;
j = ++i;          // Incrémente et renvoie le nouveau i
cout << i << j;   // Renvoie 2 2
```

Incrémentation, décrémentation (2)

En C++, un opérateur d'incrément sur un objet de type itérateur sera défini de la manière suivante:

```
class itérateur {  
    itérateur& operator++() {    // Version PREFIXEE  
        fais_une_iteration();    // PERFORMANTE  
        return *this;  
    }  
    itérateur operator++(int) {    // Version POSTFIXEE  
        itérateur vieil_iterateur = *this; // PEU PERFORMANTE  
        fais_une_iteration();  
        return vieil_iterateur;  
    }  
}
```

ATTENTION

- Le paramètre `int` n'est **pas utilisé**, il sert uniquement au mécanisme de surcharge
- Il sera toujours préférable d'utiliser la version **préfixée** des itérateurs.

L'opérateur=

Le même signe = peut servir à l'**affectation** ou à l'**initialisation**, or ce sont deux opérations différentes:

```
complexe A = 5;    // Initialisation , appel du constructeur  
complexe B = A;    // Initialisation , appel du constructeur de copie  
complexe C;  
C = A;             // Affectation , appel de operator=
```

L'initialisation correspond à:

- Allocation mémoire (ou autres ressources)
- Affectation des membres de l'objet
- Elle est réalisée par **un constructeur**

L'affectation correspond à:

- Vérifications éventuelles de compatibilité
- Affectation des membres de l'objet
- Elle est réalisée par **un operator=**

Le quintette infernal 1/3

```
class tableau {  
public:  
    tableau(const tableau& t): taille(t.taille) {  
        buffer = malloc(t.taille * sizeof(char));  
        copie(t.buffer, taille);  
    };  
    tableau& operator=(const tableau& t): taille(t.taille) {  
        if (this !=&t) { // AUTO REFERENCE  
            free(buffer);  
            buffer = malloc(taille * sizeof(char));  
            copie(t.buffer, taille);  
            return *this;  
        }  
    }  
    tableau(const tableau&& t) noexcept:  
        taille(t.taille), buffer(t.buffer) {  
        t.taille = 0;  
        t.buffer = 0;  
    };  
    tableau& operator=(tableau&& t) noexcept {  
        if (this !=&t) { // AUTO REFERENCE  
            free(buffer);  
            taille = t.taille;  
            buffer = t.buffer;  
            t.taille = 0;  
            t.buffer = nullptr;  
            return *this;  
        }  
    }  
    ~tableau() {free buffer;};  
};
```


Le quintette infernal 2/3

```
private:
    int taille;
    char* buffer;
    void copie(const char* src, int taille) ...
};

void main() {
    tableau tab1(1000);    // Constructeur
    tableau tab2 = tab1;   // Constructeur de copie
    tableau tab3(500);
    tab3 = tab2;           // Operateur=
    tableau tab4 = tab1 + tab2; // Constructeur de déplacement
    tab3 = tab4 + tab2;     // Operateur= de déplacement
};
```

- Constructeur de copie malloc + copie
- Opérateur= free + malloc
- Constructeur de déplacement copie de pointeurs
- Opérateur de déplacement free + copie de pointeurs
- Destructeur free

Le quintette infernal 3/3

trio en C++, quintette en C++11, mais toujours infernal

Il est constitué par:

- Le constructeur de copie
- L'opérateur d'affectation
- Le constructeur de déplacement
- L'opérateur de déplacement
- Le destructeur

La règle d'or du trio/quintette infernal

- *La bonne nouvelle*: Le système fournira **une version par défaut pour ces trois fonctions**
- *la mauvaise nouvelle*: La version par défaut du système **ne convient pas toujours**, en particulier s'il y a allocation de ressources dans le constructeur
- *La mauvaise nouvelle*: Si je dois écrire l'une des trois/cinq, **je dois écrire les trois/cinq** ! En effet, si vous fournissez l'un, les deux/quatre autres ne seront pas générés !
- *La bonne nouvelle*: dans du code applicatif, **les versions par défaut sont la plupart du temps suffisantes**, surtout si on utilise des objets gestionnaires de ressources

C++ moderne: La règle du zéro

Les bonnes pratiques du C++ aujourd'hui

- On n'écrit pas ses objets gestionnaires de ressources
- On utilise les conteneurs de la stl
- En particulier, on n'utilise pas de pointeurs, seulement des `unique_ptr` ou `shared_ptr`

La VRAIE règle d'or du trio/quintette infernal: Zéro code

- *La bonne nouvelle*: Pas de trio
- *La très bonne nouvelle*: Pas de quintette
- *L'excellente nouvelle*: On laisse faire le compilateur et la stl

Conversions depuis un type de base vers une classe

Les constructeurs sont utilisés pour fournir ces conversions:

```
complexe AC = {0, 1};  
  
float CF = 3.0;  
complexe CC = CF; // Conversion implicite  
AC += CF;  
  
float BF = 2.0;  
AC += BF; // Conversion implicite
```

Conversions depuis une classe vers un type de base

Les opérateurs de conversion sont utilisés pour cela:

```
class complexe {  
    public:  
        ...  
        operator float() {return r;};  
    private:  
        ...  
};  
  
main() {  
    complexe A = (3.5,4.4);  
    float Z = (float) A; // Z contient 3.5
```

(à suivre)