

Cours5 - C++

**Classes et objets en C++ : concepts plus
avancés**

J-F. Kamp

Janvier 2025

Fonction amie d'une classe

Encapsulation des données

Dans une classe, les attributs sont privés et les accès se font par l'intermédiaire d'accesseurs et modificateurs

```
class CCercle {  
  
    // D'abord les attributs d'instance privés  
    private :  
        int m_x;  
        int m_y;  
        unsigned int m_r;  
  
    // Ensuite les méthodes publiques (d'instance)  
    public :  
        // Un modificateur  
        void setRayon ( unsigned int r );  
        // Un accesseur  
        float getSurface ();  
  
    // Et les méthodes privées (d'instance)  
    private :  
        etc...  
  
};
```

La fonction amie

Une fonction amie déclarée dans une classe peut être appelée de n'importe où, sans passer par l'instanciation d'un objet de la classe

```
class CCercle {  
  
    private :  
        int m_x;  
        int m_y;  
        unsigned int m_r;  
  
    // Ensuite les méthodes publiques  
    public :  
        // Un modificateur  
        void setRayon ( unsigned int r );  
  
    // Fonction amie  
    friend bool coincide ( CCercle c1,  
                           CCercle c2 );  
  
};
```

La fonction amie

Une fonction amie déclarée dans une classe :

- accède aux attributs privés de la classe
- n'est PAS une fonction membre de la classe (méthode d'instance) => **this**
n'existe pas pour cette fonction
- peut s'écrire n'importe où dans la classe, elle est toujours visible (même si déclarée **private**)
- à utiliser avec extrême MODERATION (les puristes en P.O.O l'interdisent)
- néanmoins, en C++, elle peut devenir presque indispensable (inexistant en Java)

Exemple de fonction amie

Ecriture d'une fonction amie dans la classe

CCercle pour vérifier la coïncidence de 2 cercles

```
#include "CCercle.h"
```

```
// Code de toutes les méthodes
```

```
// Le constructeur
```

```
CCercle :: CCercle ( int x, int y, unsigned int r ) { ... }
```

```
// Les autres méthodes
```

```
...
```

```
// Une fonction amie qui renvoie vrai si 2 cercles
```

```
// passés en paramètres coïncident par leur centres.
```

```
// L'accès aux attributs privés simplifie le codage.
```

```
bool coincide ( CCercle c1, CCercle c2 ) {
```

```
    bool ret = false;
```

```
    if ( ( c1.m_x == c2.m_x ) && ( c1.m_y == c2.m_y ) ) {  
        ret = true;
```

```
    }
```

```
    return ret;
```

```
}
```

Surcharge des opérateurs

Surcharge d'opérateurs

- Par définition, un opérateur est un symbole (+, *, /, =, <<, new, etc.) qui est traduit par le langage en une opération à effectuer sur une ou des opérandes adjacentes. Ces opérateurs sont d'office définis par le langage pour des opérandes de type primitifs (bool, int, float, etc.)
- En C++, surcharger un opérateur c'est définir pour un type objet en quoi consiste l'opération pour cet objet (inexistant en Java)

Exemple :

```
CCercle c1(0, 1, 12), c2(2, 4, 25), c3;  
c3 = c1 + c2;
```

L'opération **c1 + c2** n'existe pas pour le type **CCercle** => cette opération doit être définie par une fonction (amie ou membre) pour ce type

Définition (hypothèse) : additionner 2 cercles consiste à additionner les centres et les rayons

Surcharge d'opérateurs avec une fonction amie

```
class CCercle {  
  
    private :  
        int m_x;  
        int m_y;  
        unsigned int m_r;  
  
    // Ensuite les méthodes publiques  
    public :  
        // Constructeurs  
        CCercle ( int x, int y, unsigned int r ) ;  
        CCercle () ;  
        // Un modificateur  
        void setRayon ( unsigned int r );  
        ...  
  
        // Fonction amie qui surcharge l'opérateur +  
        // Prend en paramètres les 2 cercles à additionner  
        // Renvoie un cercle = somme des 2 cercles  
        friend CCercle operator+ ( CCercle c1,  
                                    CCercle c2 );  
  
};
```

Surcharge d'opérateurs avec une fonction amie

Code de la fonction amie **operator+**

```
#include "CCercle.h"

// Code de toutes les méthodes

// Les constructeurs
CCercle :: CCercle ( int x, int y, unsigned int r ) { ... }
CCercle :: CCercle () { ... }

// Les autres méthodes
...
// La fonction amie qui surcharge l'opérateur +
CCercle operator+ (CCercle c1, CCercle c2) {

    CCercle ret;      // on suppose que le constructeur
                      // CCercle() existe dans la classe
    ret.m_x = c1.m_x + c2.m_x;
    ret.m_y = c1.m_y + c2.m_y;
    ret.m_r = c1.m_r + c2.m_r;

    return ret;
}
```

Surcharge d'opérateurs avec une fonction amie

- Maintenant, l'addition de 2 cercles est possible :
CCercle c1(0, 1, 12), c2(2, 4, 25), c3;
c3 = c1 + c2;
nouveaux attributs de c3 :
m_x vaut 2
m_y vaut 5
m_r vaut 37
- !! l'opérateur = existe par défaut pour le type CCercle mais il pourrait aussi être surchargé
- Quand le compilateur rencontre l'opération c3 = c1 + c2, grâce à la fonction amie, il transforme cette expression en :
c3 = operator+ (c1, c2);
- !! le cercle ret retourné par la fonction amie est RECOPIÉ dans c3 (ça serait une erreur de retourner CCercle& ou CCercle*)

Surcharge de l'opérateur <<

- L'opérateur << permet de transmettre la valeur d'une expression (opérande à droite de <<) dans un flot (opérande à gauche de <<)
- Dans l'expression `cout << maChaine` :
 - `cout` est le flot de sortie à l'écran, c'est un objet instance de la classe `ostream`
 - `maChaine` est une expression qui contient une chaîne de caractères (par exemple)
- L'opérateur << est défini pour tous les types primitifs (int, char, float etc.) et types chaînes de caractères
- ?? Comment surcharger l'opérateur << pour qu'il s'applique à un type `CCercle` ??

Surcharge de l'opérateur << par fonction amie

- Soit

```
CCercle c1 ( 1, 3, 15 );  
cout << c1;
```

- Hypothèse → afficher un cercle, c'est afficher ses attributs sous la forme (par ex.) :

centre en X: ...

centre en Y: ...

rayon : ...

- Ecriture de la fonction amie :

- opérateur : <<
- première opérande : **cout** type **ostream**
- deuxième opérande : **c1** type **CCercle**
- retourne : le MEME objet **cout** (dans le cas où **cout << c1 << ...**)

Surcharge de l'opérateur << par fonction amie

- signature de la fonction amie à écrire dans la classe **CCercle** :

```
friend ostream& operator<< ( ostream& os,  
                             CCercle c );
```

- code de la fonction amie :

```
#include "CCercle.h"
```

```
...
```

```
// La fonction amie qui surcharge l'opérateur <<  
ostream& operator<< (ostream& os, CCercle c)  
{
```

```
    os << "centre en X\t:" << c.m_x << endl;
```

```
    os << "centre en Y\t:" << c.m_y << endl;
```

```
    os << "rayon\t:" << c.m_r << endl;
```

```
    return os;
```

```
}
```

Héritage

Héritage simple

- Comme en Java, l'héritage entre classes est possible
- Soit la classe **de base CCercle**

```
class CCercle {  
  
    private :  
        int m_x;  
        int m_y;  
        unsigned int m_r;  
  
    // Ensuite les méthodes publiques (d'instance)  
    public :  
        // Un Constructeur  
        CCercle ( int x, int y, unsigned int r );  
        // Un modificateur  
        void setRayon ( unsigned int r );  
        // Un accesseur  
        float getSurface ();  
  
    protected :  
        void afficher ();  
  
};
```

- Comme en Java : la visibilité **protected** dans la classe de base permet d'hériter avec usufruit

Héritage simple

- Une classe **dérivée** (sous-classe) de **CCercle** pourrait être

```
#include "CCercle.h"
class CCercleTrait : public CCercle {

    private :
        // on rajoute une épaisseur de trait
        short m_ep;

    public :
        // Un modificateur
        void setEpaiss ( short ep );
};
```

- Différence avec Java : le mot-clé **public**, **protected**, **private** **CCercle** permet de modifier la visibilité dans **CCercleTrait** des champs hérités de **CCercle**

Héritage simple : redéfinition

Comme en Java, il est possible de redéfinir des méthodes de la classe de base dans la classe dérivée

```
class CCercle {  
    private :  
        ...  
  
    public :  
        ...  
        // Une méthode d'affichage  
        void afficher ();  
};
```

```
#include "CCercle.h"  
class CCercleTrait : public CCercle {  
    private :  
        short m_ep;  
  
    public :  
        ...  
        // Redéfinition de la méthode d'affichage  
        // définie dans la classe CCercle  
        void afficher ();  
};
```

Héritage simple : redéfinition

```
#include "CCercleTrait.h"

// Code des méthodes de la classe CCercleTrait

...

void CCercleTrait :: afficher () {

    // Il est parfaitement possible de d'abord appeler
    // la méthode afficher() de la classe de base CCercle
    CCercle :: afficher ();

    // Puis d'afficher l'épaisseur du trait
    cout << "L'épaisseur du trait est : " << m_ep << endl;
}
```

L'appel de l'une ou l'autre méthode **afficher()** sera simplement décidé par le type de l'objet (**CCercle** ou **CCercleTrait**)

Héritage simple : constructeurs et destructeurs

Comme en Java (mais à l'aide de l'instruction **super(...)**), il est possible de passer des paramètres au constructeur de la super-classe

```
class CCercle {  
    ...  
    public :  
        // Un constructeur  
        CCercle ( int x, int y, unsigned int r );  
    ...  
};
```

```
#include "CCercle.h"  
  
class CCercleTrait : public CCercle {  
    ...  
    public :  
        // Un constructeur  
        // Ce constructeur est également chargé de  
        // l'initialisation des attributs de CCercle  
        CCercleTrait (int x, int y, unsigned int r, short e);  
};
```

Héritage simple : constructeurs et destructeurs

```
#include "CCercleTrait.h"

// Code des méthodes de la classe CCercleTrait

...
// Constructeur de la classe CCercleTrait
// Appel du constructeur de la classe CCercle
CCercleTrait :: CCercleTrait (int x, int y, unsigned int r,
short e ) : CCercle ( x, y, r ) {

    // Initialisation uniquement de l'attribut m_ep
    m_ep = e;
}
```

Pour les destructeurs, la question ne se pose pas car ils ne prennent jamais d'arguments

Fonction virtuelle et typage dynamique

Typage statique

- Soit les objets suivants :

CCercle* c1;

CCercleTrait c2 (0, 0, 25, 4);

- Sachant que **CCercleTrait** hérite de **CCercle**, et que les 2 classes possèdent la méthode **afficher()**

- l'instruction :

c1 = &c2;

est possible et signifie que **c1** pointe maintenant sur un espace mémoire contenant des données de type **CCercleTrait**

- l'instruction : **c1->afficher()** appelle la méthode **afficher()** de la classe **CCercle** alors qu'on voudrait logiquement appeler la méthode **afficher()** de la classe **CCercleTrait**
=> ce problème vient du typage figé une fois pour toute à la compilation (typage statique)

Fonction membre virtuelle

```
class CCercle {  
    ...  
    public :  
        ...  
        // Méthode virtuelle dans la classe de base  
        virtual void afficher ();  
};
```

```
#include "CCercle.h"  
  
class CCercleTrait : public CCercle {  
    ...  
    public :  
        ...  
        // Redéfinition de la méthode afficher() dans  
        // la classe dérivée  
        virtual void afficher ();  
};
```


Fonction membre virtuelle

Dans l'exemple :

```
CCercle* c1;
```

```
CCercleTrait c2 ( 0, 0, 25, 4 );
```

```
// Cette affectation légale fait pointer c1
```

```
// sur un objet de type CCercleTrait
```

```
c1 = &c2; (1)
```

```
c1->afficher();
```

```
// afficher() étant déclarée virtuelle, le
```

```
// compilateur ne décide pas de la
```

```
// méthode appelée CCercle::afficher()
```

```
// ou CCercleTrait::afficher()
```

```
// A l'exécution, à cause de l'affectation
```

```
// (1), le type est fixé (CCercleTrait) =>
```

```
// la méthode afficher() de la classe
```

```
// CCercleTrait est exécutée
```

Fonction membre virtuelle

- Une fonction virtuelle est forcément une fonction membre (car appelée par un objet)
- Il n'y a pas grand intérêt (sauf précaution) à déclarer une fonction **virtual** si elle n'est pas redéfinie dans une classe dérivée
- Une fonction d'une classe dérivée qui redéfinit une fonction virtuelle est elle-même virtuelle
- En Java une méthode d'instance est toujours virtuelle
- En C++, la syntaxe

virtual <type_ret> maFct (...) = 0;

signifie « méthode abstraite » et est équivalent au mot-clé **abstract** de Java

C++/Java : vocabulaire identique ayant la même signification

C++/Java

constructeur

modificateur

accesseur

this

private

public

protected

new

surcharger

redéfinir

static

C++/Java : vocabulaire différent ayant la même signification

C++	Java
donnée membre	attribut
fonction membre	méthode
classe de base	super-classe
classe dérivée	sous-classe
virtual ... = 0;	abstract
const	final
pointeur	référence

C++/Java : vocabulaire C++ qui n'existe pas en Java

C++

destructeur

fonction amie

surcharge d'opérateur

copy-constructeur

virtual

passage par référence

dérivation avec visibilité