Langage C++ et programmation orientée objet

Vincent Vidal

Maître de Conférences
Enseignements : IUT Lyon 1 - pôle AP - Licence RESIR - bureau 2ème étage
Recherche : Laboratoire LIRIS - bât. Nautibus - bureau 241
E-mail : vincent.vidal@univ-lyon1.fr
Supports de cours et TPs : `https://claroline connect.univ-lyon1.fr/` espace d'activité "M41L02C - Langage C++"
26H prévues ≈ 24H de cours+TDs/TPs, et 2H - examen final
Évaluation : Contrôle continu (TPs) + examen final

Plan

Plan

Accessibilité des attributs privés d'une classe

L'encapsulation des données a la conséquence suivante : *Les attributs privés d'une classe ne sont accessibles qu'aux méthodes de leur classe*.

Seuls les membres publics sont accessibles depuis l'extérieur.

L'unité de l'encapsulation est la classe : une méthode d'une classe peut accéder aux membres privés de toutes les instances de sa classe.

Il est interdit à une fonction membre d'une classe d'accéder à des attributs privés d'une autre classe.

Accessibilité des attributs privés d'une classe

L'encapsulation des données a la conséquence suivante : *Les attributs privés d'une classe ne sont accessibles qu'aux méthodes de leur classe*.
Seuls les membres publics sont accessibles depuis l'extérieur.

L'unité de l'encapsulation est la classe : une méthode d'une classe peut accéder aux membres privés de toutes les instances de sa classe.

Il est interdit à une fonction membre d'une classe d'accéder à des attributs privés d'une autre classe.

Accessibilité des attributs privés d'une classe

L'encapsulation des données a la conséquence suivante : *Les attributs privés d'une classe ne sont accessibles qu'aux méthodes de leur classe*.
Seuls les membres publics sont accessibles depuis l'extérieur.

L'unité de l'encapsulation est la classe : une méthode d'une classe peut accéder aux membres privés de toutes les instances de sa classe.

Il est interdit à une fonction membre d'une classe d'accéder à des attributs privés d'une autre classe.

Accessibilité des attributs privés d'une classe

L'encapsulation des données a la conséquence suivante : *Les attributs privés d'une classe ne sont accessibles qu'aux méthodes de leur classe*.
Seuls les membres publics sont accessibles depuis l'extérieur.

L'unité de l'encapsulation est la classe : une méthode d'une classe peut accéder aux membres privés de toutes les instances de sa classe.

Il est interdit à une fonction membre d'une classe d'accéder à des attributs privés d'une autre classe.

Accessibilité des attributs privés d'une classe

Il est interdit à une fonction membre d'une classe d'accéder à des attributs privés d'une autre classe.

- *Si on a une classe* vecteur *et une classe* matrice, *comment définir le produit d'une matrice par un vecteur ?*

  Si la fonction que l'on veut écrire est déclarée dans une des 2 classes, elle ne pourra pas accéder aux attributs privés de l'autre classe.

- Si on a une classe Banque et une classe Compte, on aimerait faciliter l'accès aux champs privés d'un compte pour une banque...

Accessibilité des attributs privés d'une classe

Il est interdit à une fonction membre d'une classe d'accéder à des attributs privés d'une autre classe.

- *Si on a une classe* vecteur *et une classe* matrice, *comment définir le produit d'une matrice par un vecteur ?*

  Si la fonction que l'on veut écrire est déclarée dans une des 2 classes, elle ne pourra pas accéder aux attributs privés de l'autre classe.

- Si on a une classe Banque et une classe Compte, on aimerait faciliter l'accès aux champs privés d'un compte pour une banque...

Accessibilité des attributs privés d'une classe

Il est interdit à une fonction membre d'une classe d'accéder à des attributs privés d'une autre classe.

- *Si on a une classe* vecteur *et une classe* matrice, *comment définir le produit d'une matrice par un vecteur ?*

  Si la fonction que l'on veut écrire est déclarée dans une des 2 classes, elle ne pourra pas accéder aux attributs privés de l'autre classe.

- Si on a une classe Banque et une classe Compte, on aimerait faciliter l'accès aux champs privés d'un compte pour une banque...

Accessibilité des attributs privés d'une classe

Il est interdit à une fonction membre d'une classe d'accéder à des attributs privés d'une autre classe.

- *Si on a une classe* vecteur *et une classe* matrice, *comment définir le produit d'une matrice par un vecteur ?*

  Si la fonction que l'on veut écrire est déclarée dans une des 2 classes, elle ne pourra pas accéder aux attributs privés de l'autre classe.
- Si on a une classe Banque et une classe Compte, on aimerait faciliter l'accès aux champs privés d'un compte pour une banque...

Solutions ?

- Définir des **accesseurs** publics pour accéder aux membres privées :
  OK, mais si une telle méthode publique ne peut pas être "en ligne" (`inline`), alors *on sera pénalisé en temps d'exécution*.
  De plus, *l'accès public à tout peut être un problème* (e.g. numéro de compte).
- La notion de **fonction amie** (*friend* en anglais) :
  il est possible au sein d'une classe de déclarer une fonction comme amie de la classe.
  Et une même fonction peut être l'amie de plusieurs classes !

Solutions ?

- Définir des **accesseurs** publics pour accéder aux membres privées :
  OK, mais si une telle méthode publique ne peut pas être "en ligne" (**inline**), alors *on sera pénalisé en temps d'exécution*.
  De plus, *l'accès public à tout peut être un problème* (e.g. numéro de compte).
- La notion de **fonction amie** (*friend* en anglais) :
  il est possible au sein d'une classe de déclarer une fonction comme amie de la classe.
  Et une même fonction peut être l'amie de plusieurs classes !

Solutions ?

- Définir des **accesseurs** publics pour accéder aux membres privées :
  OK, mais si une telle méthode publique ne peut pas être "en ligne" (**inline**), alors *on sera pénalisé en temps d'exécution*.
  De plus, *l'accès public à tout peut être un problème* (e.g. numéro de compte).
- La notion de **fonction amie** (*friend* en anglais) :
  il est possible au sein d'une classe de déclarer une fonction comme amie de la classe.
  Et une même fonction peut être l'amie de plusieurs classes !

Solutions ?

- Définir des **accesseurs** publics pour accéder aux membres privées :
  OK, mais si une telle méthode publique ne peut pas être "en ligne" (**inline**), alors *on sera pénalisé en temps d'exécution*.
  De plus, *l'accès public à tout peut être un problème* (e.g. numéro de compte).
- La notion de **fonction amie** (*friend* en anglais) :
  il est possible au sein d'une classe de déclarer une fonction comme amie de la classe.
  Et une même fonction peut être l'amie de plusieurs classes !

Solutions ?

- Définir des **accesseurs** publics pour accéder aux membres privées :
  OK, mais si une telle méthode publique ne peut pas être "en ligne" (**inline**), alors *on sera pénalisé en temps d'exécution*.
  De plus, *l'accès public à tout peut être un problème* (e.g. numéro de compte).
- La notion de **fonction amie** (*friend* en anglais) :
  il est possible au sein d'une classe de déclarer une fonction comme amie de la classe.
  Et une même fonction peut être l'amie de plusieurs classes !

```
class point2D
{
  private :
    float m_x, m_y ;
  public :
    point2D(float x=0.f, float y=0.f) { m_x = x; m_y = y; }    // constructeur "en ligne"
    void affiche() ;                                           // méthode de la classe

    friend bool est_identique(const point2D&, const point2D&); // fonction indépendante amie
                                                               // de la classe point2D
};

void point2D::affiche(){ std::cout << " (" << m_x << ", " << m_y << ")" << std::endl; }

// est_identique n'est pas une fonction membre de point2D (donc aucune influence des
// mots-clés public ou private)
// l'amitié permet d'avoir des fonctions dont l'implantation est "symétrique"
bool est_identique(const point2D& p1, const point2D& p2){
    return (p1.m_x==p2.m_x && p1.m_y==p2.m_y);
}

int main(){
    point2D p, pbis;
    if(est_identique(p, pbis))
        std::cout << " Les 2 points 2D sont identiques. " << std::endl;
    return 0;
}
```

Fonction membre d'une classe amie d'une autre classe

```
class B
{
  ... // données privées
  public :
  ...
    void methode(const A&);           // méthode de B qui doit accéder aux membres privés de A
};

class A
{
  ... // données privées
  public :
  ...
    friend void B::methode(const A&); // j'autorise la méthode "methode" de la classe B à
                                      // accéder aux membres privés de A
};
```

La classe B doit être déclarée avant la classe A pour que je puisse déclarer la méthode methode de B comme amie de A. **Mais** pour compiler la déclaration de la méthode methode au sein de B, le compilateur a besoin de savoir que A est une classe.

```
class B
{
  ... // données privées
  public :
  ...
    void methode(const A&);           // méthode de B qui doit accéder aux membres privés de A
};

class A
{
  ... // données privées
  public :
  ...
    friend void B::methode(const A&); // j'autorise la méthode "methode" de la classe B à
                                      // accéder aux membres privés de A
};
```

La classe B doit être déclarée avant la classe A pour que je puisse déclarer la méthode methode de B comme amie de A.

Mais pour compiler la déclaration de la méthode methode au sein de B, le compilateur a besoin de savoir que A est une classe.

```
class B
{
  ... // données privées
  public :
  ...
    void methode(const A&);           // méthode de B qui doit accéder aux membres privés de A
};

class A
{
  ... // données privées
  public :
  ...
    friend void B::methode(const A&); // j'autorise la méthode "methode" de la classe B à
                                      // accéder aux membres privés de A
};
```

La classe B doit être déclarée avant la classe A pour que je puisse déclarer la méthode methode de B comme amie de A. **Mais** pour compiler la déclaration de la méthode methode au sein de B, le compilateur a besoin de savoir que A est une classe.

Solution :

```
class A;                                  // on dit au compilateur que A est une classe

class B {
  ... // données privées
  public :
  ...
    void methode(const A&);               // méthode qui doit accéder aux membres privés de A
};

class A {
  ... // données privées
  public :
  ...
    friend void B::methode(const A&); // j'autorise la méthode "methode" de B à
                                      // accéder aux membres privés de A
};

void B::methode(const A& a) { ... }   // la définition de methode nécessite la connaissance de A!
```

Si les classes A et B sont dans 2 composants logiciels différents (.hpp/.cpp), alors le fichier d'en-tête qui contient la déclaration de A doit inclure le fichier d'en-tête contenant la déclaration de B.

Solution :

```
class A;                                  // on dit au compilateur que A est une classe

class B {
  ... // données privées
  public :
  ...
    void methode(const A&);               // méthode qui doit accéder aux membres privés de A
};

class A {
  ... // données privées
  public :
  ...
    friend void B::methode(const A&); // j'autorise la méthode "methode" de B à
                                      // accéder aux membres privés de A
};

void B::methode(const A& a) { ... }   // la définition de methode nécessite la connaissance de A!
```

Si les classes A et B sont dans 2 composants logiciels différents (.hpp/.cpp), alors le fichier d'en-tête qui contient la déclaration de A doit inclure le fichier d'en-tête contenant la déclaration de B.

Le cas d'amitiés croisées entre fonctions de 2 classes est plus compliqué, puisqu'il faudra déclarer au moins une des 2 classes comme amie de l'autre. Cela se fait par exemple en écrivant `friend class A;` au début de la déclaration de la classe B.

Classe amie d'une autre classe

```
class B
{
    friend class A ; // toutes les méthodes de A sont amies de B
  ... // données privées
  public :
  ...
};

class A
{
  ... // données privées
  public :
  ...
};
```

Déclarer qu'une classe A est amie d'une classe B signifie que toutes les méthodes de A sont amies de B.

Attention : la relation d'amitié n'est ni réciproque, ni transitive !

```
class B
{
    friend class A ; // toutes les méthodes de A sont amies de B
  ... // données privées
  public :
  ...
};

class A
{
  ... // données privées
  public :
  ...
};
```

Déclarer qu'une classe A est amie d'une classe B signifie que toutes les méthodes de A sont amies de B.

Attention : la relation d'amitié n'est ni réciproque, ni transitive !

```
class B
{
    friend class A ; // toutes les méthodes de A sont amies de B
  ... // données privées
  public :
  ...
};

class A
{
  ... // données privées
  public :
  ...
};
```

Déclarer qu'une classe A est amie d'une classe B signifie que toutes les méthodes de A sont amies de B.

Attention : la relation d'amitié n'est ni réciproque, ni transitive !

Application

```
class Compte
{
    std::string m_nomCompte ;
    float m_soldeCompte ;
    float m_decouvertMax ;
    friend class Banque; // toutes les méthodes de Banque sont amies de Compte
                         // elles peuvent donc accéder directement aux membres
                         // privés d'un objet de type Compte
  public : ...
};
class Banque
{
    Compte* m_ad_debut ;
    int m_nbct, m_nbc ;
  public :
    Banque(int nbct=10){ m_ad_debut=new Compte[m_nbct=nbct]; m_nbc=0; }
    void ajouter(std::string, float, float) ;
};
void Banque::ajouter(std::string nomCompte, float soldeCompte, float decouvertMax) {
    if( m_nbc == m_nbct) ... ;
    m_ad_debut[m_nbc].m_nomCompte = nomCompte ;
    m_ad_debut[m_nbc].m_soldeCompte = soldeCompte ;
    m_ad_debut[m_nbc].m_decouvertMax = decouvertMax ;
    m_nbc++ ; }
```

Ce qu'il faut retenir sur l'amitié

Intérêts :

- Simplifier la syntaxe et accélérer le temps d'accès aux membres privés pour quelques fonctions et méthodes privilégiées.
- Ne pas détruire (totalement) l'encapsulation des données (ce qui se produit si on rend les données publiques).
- Obtenir des implantions symétriques pour les fonctions de comparaison.

Ce qu'il faut retenir sur l'amitié

Intérêts :

- Simplifier la syntaxe et accélérer le temps d'accès aux membres privés pour quelques fonctions et méthodes privilégiées.
- Ne pas détruire (totalement) l'encapsulation des données (ce qui se produit si on rend les données publiques).
- Obtenir des implantions symétriques pour les fonctions de comparaison.

Ce qu'il faut retenir sur l'amitié

Intérêts :

- Simplifier la syntaxe et accélérer le temps d'accès aux membres privés pour quelques fonctions et méthodes privilégiées.
- Ne pas détruire (totalement) l'encapsulation des données (ce qui se produit si on rend les données publiques).
- Obtenir des implantions symétriques pour les fonctions de comparaison.

Plan

Généralités

Le C++ autorise la surcharge (ou surdéfinition) de fonctions, qu'il s'agisse de méthodes ou de fonctions indépendantes.

Le C++ permet également de surdéfinir n'importe quel opérateur unaire et binaire. *Des exemples d'opérateurs en tête ?*

Généralités

Le C++ autorise la surcharge (ou surdéfinition) de fonctions, qu'il s'agisse de méthodes ou de fonctions indépendantes.

Le C++ permet également de surdéfinir n'importe quel opérateur unaire et binaire. *Des exemples d'opérateurs en tête ?*

Généralités

Le C++ autorise la surcharge (ou surdéfinition) de fonctions, qu'il s'agisse de méthodes ou de fonctions indépendantes.

Le C++ permet également de surdéfinir n'importe quel opérateur unaire et binaire. *Des exemples d'opérateurs en tête ?*

- **Opérateurs binaires** : l'affectation =, l'addition +, la soustraction -, la multiplication *, les opérateurs relationnels (<, <=, >, >=, ==, !=)...
- **Opérateurs unaires** : l'addition +, la soustraction -, le déréférencement *, **new**, **delete**...

Généralités

Le C++ autorise la surcharge (ou surdéfinition) de fonctions, qu'il s'agisse de méthodes ou de fonctions indépendantes.

Le C++ permet également de surdéfinir n'importe quel opérateur unaire et binaire. *Des exemples d'opérateurs en tête ?*

- **Opérateurs binaires** : l'affectation =, l'addition +, la soustraction -, la multiplication *, les opérateurs relationnels (<, <=, >, >=, ==, !=)...
- **Opérateurs unaires** : l'addition +, la soustraction -, le déréférencement *, **new**, **delete**...

C++ surdéfinit déjà les opérateurs pour les types de base (**int**, **double**,...) et vous n'avez donc pas le droit de le faire également.

Intérêt ?

On a déjà des méthodes qui peuvent faire toutes les opérations nécessaires, *alors pourquoi préférer l'utilisation d'opérateurs ?*

La notation opératoire est beaucoup plus concise et lisible qu'une notation fonctionnelle.

Exemple : Si on a 2 objets de type `Point2D`, p1 et p2, alors `p1 == p2` est plus facile à lire que `p1.est_identique(p2)` ou que `est_identique(p1, p2)` si est_identique est une fonction amie de la classe `Point2D`.

Intérêt ?

On a déjà des méthodes qui peuvent faire toutes les opérations nécessaires, *alors pourquoi préférer l'utilisation d'opérateurs ?*

La notation opératoire est beaucoup plus concise et lisible qu'une notation fonctionnelle.

Exemple : Si on a 2 objets de type `Point2D`, `p1` et `p2`, alors `p1 == p2` est plus facile à lire que `p1.est_identique(p2)` ou que `est_identique(p1, p2)` si `est_identique` est une fonction amie de la classe `Point2D`.

Intérêt ?

On a déjà des méthodes qui peuvent faire toutes les opérations nécessaires, *alors pourquoi préférer l'utilisation d'opérateurs ?*

La notation opératoire est beaucoup plus concise et lisible qu'une notation fonctionnelle.

Exemple : Si on a 2 objets de type Point2D, p1 et p2, alors p1 == p2 est plus facile à lire que p1.est_identique(p2) ou que est_identique(p1, p2) si est_identique est une fonction amie de la classe Point2D.

p1+p2 où p1 et p2 de type Point2D

Mot-clé *operator* suivi de l'opérateur +.

```
#include <iostream>
class Point2D {
    float m_x, m_y ;
  public :
    Point2D(float x=0, float y=0): m_x(x),m_y(y){ cout << "Construc. usuel" << endl ; }
    Point2D(const Point2D& p): m_x(p.m_x),m_y(p.m_y){ cout << "Construc. par recopie" << endl ; }
    void affiche() const { cout << "(" << m_x << ", " << m_y << ")" << endl; }
    // opérateur + comme fonction amie :
    friend Point2D operator + (const Point2D&, const Point2D&) ;
};

Point2D operator + (const Point2D& p1, const Point2D& p2){
    Point2D p(p1.m_x+p2.m_x, p1.m_y+p2.m_y) ;
    return p ;
}

int main(){
    Point2D a(5,-8), b(3,7), c ;
    c = a+b ;
    c.affiche() ;
    (a+b+c).affiche() ;
    return EXIT_SUCCESS ;
}
```

p1+p2 sans l'amitié

Mot-clé *operator* suivi de l'opérateur +.

```
#include <iostream>
class Point2D {
    float m_x, m_y ;
  public :
    Point2D(float x=0, float y=0): m_x(x),m_y(y){ cout << "Construc. usuel" << endl ; }
    Point2D(const Point2D& p): m_x(p.m_x),m_y(p.m_y){ cout << "Construc. par recopie" << endl ; }
    void affiche() const { cout << "(" << m_x << ", " << m_y << ")" << endl; }
    // opérateur + comme fonction membre :
    Point2D operator + (const Point2D&) const ; // le 1er paramètre de l'addition est implicite
};

Point2D Point2D::operator + (const Point2D& p) const {
    Point2D ptmp(m_x+p.m_x, m_y+p.m_y) ; // sans l'amitié, une dissymétrie apparaît
    return ptmp ;
}

int main(){
    Point2D a(5,-8), b(3,7), c ;
    c = a+b ; // equivalent à a.operator+(b)
    c.affiche() ;
    (a+b+c).affiche() ;
    return EXIT_SUCCESS ;
}
```

Choisir entre un opérateur membre ou ami ?

- Type_C++ $\otimes$ Classe : **forcément un opérateur $\otimes$ ami** ; Type_C++ = Type_scalaire ou classe définis par C++ (e.g. dans std) ;
- Classe $\otimes$ Type_C++ : les 2 conviennent ;
- Classe $\otimes$ Classe : les 2 conviennent.
- Classe1 $\otimes$ Classe2 : les 2 conviennent.

Il est conseillé de définir les opérateurs qui vont modifier un objet comme fonction membre (pour éviter des problèmes de conversion implicite liés aux types dérivés).

Choisir entre un opérateur membre ou ami ?

- Type_C++ $\otimes$ Classe : **forcément un opérateur $\otimes$ ami** ; Type_C++ = Type_scalaire ou classe définis par C++ (e.g. dans std) ;
- Classe $\otimes$ Type_C++ : les 2 conviennent ;
- Classe $\otimes$ Classe : les 2 conviennent.
- Classe1 $\otimes$ Classe2 : les 2 conviennent.

Il est conseillé de définir les opérateurs qui vont modifier un objet comme fonction membre (pour éviter des problèmes de conversion implicite liés aux types dérivés).

Choisir entre un opérateur membre ou ami ?

- Type_C++ $\otimes$ Classe : **forcément un opérateur $\otimes$ ami** ; Type_C++ = Type_scalaire ou classe définis par C++ (e.g. dans std) ;
- Classe $\otimes$ Type_C++ : les 2 conviennent ;
- Classe $\otimes$ Classe : les 2 conviennent.
- Classe1 $\otimes$ Classe2 : les 2 conviennent.

Il est conseillé de définir les opérateurs qui vont modifier un objet comme fonction membre (pour éviter des problèmes de conversion implicite liés aux types dérivés).

Choisir entre un opérateur membre ou ami ?

- Type_C++ $\otimes$ Classe : **forcément un opérateur $\otimes$ ami** ; Type_C++ = Type_scalaire ou classe définis par C++ (e.g. dans std) ;
- Classe $\otimes$ Type_C++ : les 2 conviennent ;
- Classe $\otimes$ Classe : les 2 conviennent.
- Classe1 $\otimes$ Classe2 : les 2 conviennent.

Il est conseillé de définir les opérateurs qui vont modifier un objet comme fonction membre (pour éviter des problèmes de conversion implicite liés aux types dérivés).

Choisir entre un opérateur membre ou ami ?

- Type_C++ $\otimes$ Classe : **forcément un opérateur $\otimes$ ami** ; Type_C++ = Type_scalaire ou classe définis par C++ (e.g. dans std) ;
- Classe $\otimes$ Type_C++ : les 2 conviennent ;
- Classe $\otimes$ Classe : les 2 conviennent.
- Classe1 $\otimes$ Classe2 : les 2 conviennent.

Il est conseillé de définir les opérateurs qui vont modifier un objet comme fonction membre (pour éviter des problèmes de conversion implicite liés aux types dérivés).

Règles

En C++, vous pouvez surdéfinir un opérateur s'il porte sur *au moins 1 objet* (1 instance d'une classe) et *si cet opérateur fait partie de la liste des opérateurs C++*.

Il n'est pas possible de créer un nouveau symbole pour une opération, on peut utiliser seulement certains des opérateurs surdéfinis pour les types de base.

Règles

En C++, vous pouvez surdéfinir un opérateur s'il porte sur *au moins 1 objet* (1 instance d'une classe) et *si cet opérateur fait partie de la liste des opérateurs C++*.

Il n'est pas possible de créer un nouveau symbole pour une opération, on peut utiliser seulement certains des opérateurs surdéfinis pour les types de base.

Règles

Il faut **conserver la pluralité (unaire ou binaire) de l'opérateur**. Par exemple, l'opérateur ++ ne peut être surdéfini que dans le cas unaire et l'opérateur = ne peut être défini que dans le cas binaire.

Les **priorités relatives des opérateurs et les règles d'associativité pour les types de base sont conservées** pour les opérateurs surdéfinis.

Règles

Il faut **conserver la pluralité (unaire ou binaire) de l'opérateur**. Par exemple, l'opérateur ++ ne peut être surdéfini que dans le cas unaire et l'opérateur = ne peut être défini que dans le cas binaire.

Les **priorités relatives des opérateurs et les règles d'associativité pour les types de base sont conservées** pour les opérateurs surdéfinis.

Tableau des opérateurs surdéfinissables par priorité

1. **référence (GD)** () [] -> *Attention : ces 3 opérateurs doivent être des fonctions membres*
2. **UNAIRES (DG)** + − ++ −− ! ~ * & (cast) **new new[] delete delete[]**
3. **arithmétique (GD)** * / %
4. **arithmétique (GD)** + −
5. **décalage (GD)** « »
6. **relationnels (GD)** < <= > >=
7. **relationnels (GD)** ==!=
8. **manipulation de bits (GD)** &
9. **manipulation de bits (GD)** ∧
10. **logique (GD)** ||
11. **logique (GD)** &&
12. **manipulation de bits (GD)** |
13. **affectation (DG)** = += -= *= /= %= &= ∧= |= «= »=
14. **séquentiel (GD)** ,

GD : associativité de la gauche vers la droite ;
DG : associativité de la droite vers la gauche.

Aucune commutativité des opérateurs

Soit une classe CComplexe pour représenter un nombre complexe.
On souhaite rendre possible l'addition d'un complexe et d'un flottant, en interprétant le flottant comme un complexe dont la partie imaginaire est nulle.

Nous définissons un opérateur + ami de la classe CComplexe dont l'en-tête de la définition est :
CComplexe operator + (const CComplexe& c, double d)
Si a est un objet de type CComplexe :

- a + 5.2 a du sens ;
- 5.2 + a n'a pas de sens (erreur de compilation).

Aucune commutativité des opérateurs

Soit une classe CComplexe pour représenter un nombre complexe.
On souhaite rendre possible l'addition d'un complexe et d'un flottant, en interprétant le flottant comme un complexe dont la partie imaginaire est nulle.

Nous définissons un opérateur + ami de la classe CComplexe dont l'en-tête de la définition est :
CComplexe operator + (const CComplexe& c, double d)
Si a est un objet de type CComplexe :

- a + 5.2 a du sens ;
- 5.2 + a n'a pas de sens (erreur de compilation).

Aucune commutativité des opérateurs

Soit une classe CComplexe pour représenter un nombre complexe.
On souhaite rendre possible l'addition d'un complexe et d'un flottant, en interprétant le flottant comme un complexe dont la partie imaginaire est nulle.

Nous définissons un opérateur + ami de la classe CComplexe dont l'en-tête de la définition est :
CComplexe operator + (const CComplexe& c, double d)
Si a est un objet de type CComplexe :

- a + 5.2 a du sens ;
- 5.2 + a n'a pas de sens (erreur de compilation).

Aucune commutativité des opérateurs

Soit une classe CComplexe pour représenter un nombre complexe.
On souhaite rendre possible l'addition d'un complexe et d'un flottant, en interprétant le flottant comme un complexe dont la partie imaginaire est nulle.

Nous définissons un opérateur + ami de la classe CComplexe dont l'en-tête de la définition est :
`CComplexe operator + (const CComplexe& c, double d)`
Si a est un objet de type CComplexe :

- `a + 5.2` a du sens ;
- `5.2 + a` n'a pas de sens (erreur de compilation).

Solutions ?

- **Solution 1** (non-optimale) : définir aussi un opérateur + entre un double et un complexe. L'en-tête serait :
  `CComplexe operator + (double d, const CComplexe& c) ;`
- **Solution 2** (optimale) : définir l'opérateur + seulement pour 2 nombres complexes, puis définir l'opération de conversion d'un double en CComplexe ;

  la conversion de type sera traitée par la surdéfinition d'un constructeur pour la classe CComplexe avec 1 argument de type **double**.

Solutions ?

- **Solution 1** (non-optimale) : définir aussi un opérateur + entre un double et un complexe. L'en-tête serait : `CComplexe operator + (double d, const CComplexe& c);`
- **Solution 2** (optimale) : définir l'opérateur + seulement pour 2 nombres complexes, puis définir l'opération de conversion d'un double en `CComplexe` ;

  la conversion de type sera traitée par la surdéfinition d'un constructeur pour la classe `CComplexe` avec 1 argument de type `double`.

Solutions ?

- **Solution 1** (non-optimale) : définir aussi un opérateur + entre un double et un complexe. L'en-tête serait : `CComplexe operator + (double d, const CComplexe& c);`
- **Solution 2** (optimale) : définir l'opérateur + seulement pour 2 nombres complexes, puis définir l'opération de conversion d'un double en `CComplexe` ;

  la conversion de type sera traitée par la surdéfinition d'un constructeur pour la classe `CComplexe` avec 1 argument de type **`double`**.

L'opérateur = est déjà prédéfini

L'opérateur = défini par défaut correspond à la copie des valeurs des attributs d'instance de son second opérande dans le premier.

Cela va poser des problèmes en cas d'attributs d'instance pointant sur des emplacements mémoires dynamiques, car les emplacements dynamiques seraient partagés entre plusieurs objets, ce qu'il faut éviter.

Analogie avec la problématique du constructeur par recopie, mais au-lieu d'avoir des recopies membre par membre, il y a ici une affectation membre par membre.

L'opérateur = est déjà prédéfini

L'opérateur = défini par défaut correspond à la copie des valeurs des attributs d'instance de son second opérande dans le premier.

Cela va poser des problèmes en cas d'attributs d'instance pointant sur des emplacements mémoires dynamiques, car les emplacements dynamiques seraient partagés entre plusieurs objets, ce qu'il faut éviter.

Analogie avec la problématique du constructeur par recopie, mais au-lieu d'avoir des recopies membre par membre, il y a ici une affectation membre par membre.

L'opérateur = est déjà prédéfini

L'opérateur = défini par défaut correspond à la copie des valeurs des attributs d'instance de son second opérande dans le premier.

Cela va poser des problèmes en cas d'attributs d'instance pointant sur des emplacements mémoires dynamiques, car les emplacements dynamiques seraient partagés entre plusieurs objets, ce qu'il faut éviter.

Analogie avec la problématique du constructeur par recopie, mais au-lieu d'avoir des recopies membre par membre, il y a ici une affectation membre par membre.

Exemple

```
#include <iostream>
class Point2D {
    float m_x, m_y ;
  public :
    Point2D(float x=0, float y=0): m_x(x),m_y(y){ cout << "Construc. usuel" << endl ; }
    Point2D(const Point2D& p): m_x(p.m_x),m_y(p.m_y){ cout << "Construc. par recopie" << endl ; }
    void affiche() const { cout << "(" << m_x << ", " << m_y << ")" << endl; }
    // l'opérateur = DOIT ETRE une fonction membre :
    Point2D& operator = (const Point2D&) ; // on retourne une référence pour autoriser
                                           // les affectations en série (et pour
                                           // ne pas faire appel au constructeur de recopie!)
};
Point2D& Point2D::operator = (const Point2D& p){
    if (this == &p) // On essaie d'affecter l'objet à soi-même?
      return *this; // Oui, alors on ne fait rien, on retourne simplement *this.

    // 1) libérer les emplacements dynamiques initiaux de l'objet courant
    // 2) allouer les emplacements dynamiques pour contenir les nouvelles données
    // 3) copier les valeurs à l'aide de l'opérateur d'affectation
    m_x = p.m_x;
    m_y = p.m_y;
    // 4) retourner une référence sur soi-même
    return *this; // retourne une référence sur soi-même
}
```

Exercice

A vous de jouer : reprendre la classe tableau du cours 3 (slide 26) pour représenter un tableau dynamique et

- proposer une surdéfinition de l'opérateur d'affectation : un binôme passe au tableau (bonus moyenne TP) ;
- proposer une surdéfinition de l'opérateur [] pour faciliter l'accès aux cases du tableau dynamique via leur indice ; [] a 1 argument de type `int` et une valeur de retour de type `TypeElementCase&`.

Exercice

A vous de jouer : reprendre la classe tableau du cours 3 (slide 26) pour représenter un tableau dynamique et

- proposer une surdéfinition de l'opérateur d'affectation : un binôme passe au tableau (bonus moyenne TP) ;
- proposer une surdéfinition de l'opérateur [] pour faciliter l'accès aux cases du tableau dynamique via leur indice ; [] a 1 argument de type **int** et une valeur de retour de type TypeElementCase&.

Cas général

Les opérateurs « et » sont déjà surdéfinis au sein des classes istream **et** ostream **pour les types de base**. Ces opérateurs ont une forme particulière :

- **leur 1er argument DOIT être un flot** : cela empêche de surdéfinir « et » comme fonction membre, **il faut donc forcément passer par l'amitié** ;
- la **valeur de retour EST OBLIGATOIREMENT la référence du flot** concerné (pour autoriser la mise en série des flots) ;
- ostream & operator « (ostream &, **const** type_classe &) ; ;
- istream & operator » (istream &, type_classe &) ;(ICI PAS **const**).

Cas général

Les opérateurs « et » sont déjà surdéfinis au sein des classes istream **et** ostream **pour les types de base**. Ces opérateurs ont une forme particulière :

- **leur 1er argument DOIT être un flot** : cela empêche de surdéfinir « et » comme fonction membre, **il faut donc forcément passer par l'amitié** ;
- la **valeur de retour EST OBLIGATOIREMENT la référence du flot** concerné (pour autoriser la mise en série des flots) ;
- ostream & operator « (ostream &, **const** type_classe &) ; ;
- istream & operator » (istream &, type_classe &) ;(ICI PAS **const**).

Cas général

Les opérateurs « et » sont déjà surdéfinis au sein des classes istream **et** ostream **pour les types de base**. Ces opérateurs ont une forme particulière :

- **leur 1er argument DOIT être un flot** : cela empêche de surdéfinir « et » comme fonction membre, **il faut donc forcément passer par l'amitié** ;
- la **valeur de retour EST OBLIGATOIREMENT la référence du flot** concerné (pour autoriser la mise en série des flots) ;
- ostream & operator « (ostream &, **const** type_classe &) ; ;
- istream & operator » (istream &, type_classe &) ;(ICI PAS **const**).

Cas général

Les opérateurs « et » sont déjà surdéfinis au sein des classes istream **et** ostream **pour les types de base**. Ces opérateurs ont une forme particulière :

- **leur 1er argument DOIT être un flot** : cela empêche de surdéfinir « et » comme fonction membre, **il faut donc forcément passer par l'amitié** ;
- la **valeur de retour EST OBLIGATOIREMENT la référence du flot** concerné (pour autoriser la mise en série des flots) ;
- ostream & operator « (ostream &, **const** type_classe &) ; ;
- istream & operator » (istream &, type_classe &) ;(ICI PAS **const**).

Exemple

```
#include <iostream>
using namespace std ;

class Point2D {
    float m_x, m_y ;
  public :
    Point2D(float x=0, float y=0): m_x(x),m_y(y){ cout << "Construc. usuel" << endl ; }
    Point2D(const Point2D& p): m_x(p.m_x),m_y(p.m_y){ cout << "Construc. par recopie" << endl ; }

    // l'opérateur << DOIT ETRE une fonction amie :
    friend ostream& operator << (ostream&, const Point2D&) ;
    // l'opérateur >> DOIT ETRE une fonction amie :
    friend istream& operator >> (istream&, Point2D&) ;
};

ostream& operator << (ostream& sortie, const Point2D& p){
    sortie << "(" << p.m_x << "," << p.m_y << ")" ;
    return sortie ;
}
```

Exemple (suite)

```
istream& operator >> (istream& entree, Point2D& p){
    char c = '\0' ; float x, y; bool ok = true ;
    entree >> c ; // lecture du 1er caractère de la tête de lecture
    if(c != '(') ok = false ;
    else {
      entree >> x >> c ; // on lit le 1er nombre et le caractère suivant
      if(c !=',') ok = false ;
      else {
        entree >> y >> c ; // on lit le 2ème nombre et le dernier caractère
        if(c != ')') ok = false ;
      }
    }
    if(ok){ p.m_x = x ; p.m_y = y ; }
    else entree.clear( ios::badbit | entree.rdstate() ) ; // activer le bit d'erreur sans
                                                          // activer les autres
                                                          // ainsi entree.bad() sera vrai
                                                          // (flot dans état irrécupérable)
    return entree ;
}
```

Exemple (fin)

```
int main(){
    Point2D p(2,8) ;
    cout << " Mon point initial est " << p << endl ;

    bool ok = false ;
    do
    {
        cout << "Donnez un point : " ;
        if(cin >> p){ cout << " Merci pour le point " << p << endl; ok = true ; }
        else
        {
            cout << " Information saisie incorrecte! " << endl ;
            cin.clear() ;
        }
    }while( !ok );

    return EXIT_SUCCESS ;
}
```