Algorithmique et Programmation 2 : Classes en C++ (Suite)

Opérateurs et Surcharge

- Il est possible de définir des opérations liées à une classe au moyen de la ***surcharge*** d'opérateurs existants.

- Par exemple, si on veut définir une classe Entier dotée d'une opération « + », on peut surcharger l'opérateur « + » pour la classe Entier.

Opérateurs et Surcharge

- Classe Entier

```
Entier Entier::operator+(const Entier &e) {
  Entier result;
  result.my_valeur = my_valeur + e.my_valeur;
  return result;
}
```

```
  Entier a(5);
   Entier b(6);
   cout << (a+b).getValeur() << endl;
```

Opérateurs et Surcharge

- Il est possible de définir des opérations liées à une classe au moyen de la ***surcharge*** d'opérateurs existants.
- Par exemple, si on veut définir une classe Entier dotée d'une opération « + », on peut surcharger l'opérateur « + » pour la classe Entier.
- Il est possible de surcharger la plupart des opérateurs existants: +, -, /, *, ==, !=, <, <=, >, >=, !, <<, >>, ++, --, +=, -=, ...

Opérateurs et Surcharge : Exercice

- Reprendre la classe ***Entier*** et surcharger les opérateurs classiques (+, -, *, /)
- Question complémentaire : utilité du const?

Opérateurs et Surcharge : opérateur d'affectation

- Pour une classe X, il est appelé à chaque fois que l'on écrit une affectation avec un objet de la classe X (dans la partie gauche de l'affectation)

```
X obj; // appel du constructeur par defaut de la classe X
X obj2( "chaine qcq" ); // appel du constructeur de X prenant une chaine.
obj = obj2; // appel de l'operateur d'affectation de X.
// Attention, les trois exemples suivants ne sont pas des affectations
// mais des constructions par copie.
X obj3 = obj2; // appel du constructeur par copie de X.
X obj3( obj2 ); // equivalent
X obj3 = X( obj2 ); // equivalent
```

Opérateurs et Surcharge : opérateur d'affectation

- L'***opérateur d'affectation*** (ou operator=) permet de mettre un objet existant dans le même état qu'un autre objet du même type.
- Cet opérateur est défini implicitement par le compilateur pour toutes les classes (comme le constructeur par défaut/copie)

Opérateurs et Surcharge : opérateur d'affectation

- Prototype : X operator=(const X &autre)

Opérateurs et Surcharge : opérateur d'affectation

- ***Prototype***: X & X::operator=(const X &autre)

```
// prend un autre vecteur en parametre,
// retourne soi-meme.
Vecteur&
Vecteur::operator=( const Vecteur & autre )
{ // On fait l'affectation seulement si
  // autre est un objet different de
  // soi-meme (this).
  if ( this != &autre )
    {
      my_dx = autre.my_dx;
      my_dy = autre.my_dy;
    }
  return *this;
}
```

Opérateurs et Surcharge : Exercice

- Reprendre la classe ***Entier*** et surcharger l'opérateur d'affectation

Opérateurs et Surcharge : Exercice

- Créer la classe matrice correspondant à

```
class Matrice {
  public:
    Matrice(unsigned rows, unsigned cols);
    double operator() (const unsigned row,const unsigned col);
    double  operator() (const unsigned row, const unsigned col) const;
    ~Matrice();                              // Destructeur
    Matrice(const Matrice& m);               // Constructeur de copie
    Matrice& operator= (const Matrice& m);   // Opérateur d'assignement

  private:
    unsigned rows_, cols_;
    double* data_;
};
```

Depuis le main :

Matrice a(3,2); cout << a(1,1) << endl;

Intérêt du const?

Objet passés en paramètres ou retournés

Comme toute variable, un objet peut être utilisé comme argument d'appel d'une méthode ou fonction

- Passage par ***valeur***
- Passage par ***référence***

Objet passés en paramètres ou retournés

Comme toute variable, un objet peut être utilisé comme argument d'appel d'une méthode ou fonction

- Passage par ***valeur*** : définit dans le prototype de la fonction avec un nom de classe suivi d'un nom de variable
  - Création d'un clone lors de l'appel
  - Destruction du clone à la fin de la fonction
- Passage par ***référence***

Objet passés en paramètres ou retournés

Comme toute variable, un objet peut être utilisé comme argument d'appel d'une méthode ou fonction

- Passage par ***valeur***
- Passage par ***référence*** : définit dans le prototype de la fonction avec un nom de classe suivi du symbole « & » suivi d'un nom de variable
  - Pas de copie, utilisation d'un synonyme de l'objet
  - Pas de destruction à la fin de la fonction

Objet passés en paramètres ou retournés

- Passage par ***pointeur*** : est un type particulier de passage par valeur dont la valeur copiée est l'adresse de l'objet
- Passage par ***référence constante*** est un type particulier de passage par référence dont le paramètre ne peut être modifié (accès en lecture uniquement)

Objet passés en paramètres ou retournés

- Un objet peut être ***retourné*** à la fin d'une fonction/méthode (si celle çi définit dans son prototype une classe comme valeur de retour).
- Le constructeur par copie est appelé et la valeur de retour est un clone de l'objet fabriqué dans la fonction et retourné à la fin de celle-çi

Objet passés retournés : Exercice

- Sur l'exemple suivant, combien d'instances de Vecteur sont créés ? Précisez pour chaque instance où elle est crée/détruite.

```
bool estColineaire( Vecteur u, const Vecteur & v )
{
  return ( u.x() * v.y() - u.y() * v.x() ) == 0;
}

Vecteur orthogonal( const Vecteur & u )
{
  Vecteur n( -u.y(), u.x() );
  return n;
}

void main()
{
  Vecteur a( 3, 2 );
  Vecteur b;
  b = a.orthogonal();
  if ( estColineaire( a, b ) )
    cerr << "Probleme !" << endl;
}
```

Méthodes constantes

- Une méthode dont le prototype est terminé par **const** est appelée ***méthode constante***.
- Dans le corps d'une méthode constante, il est impossible de modifier les attributs de l'objet et d'appeler des méthodes non-constantes de cet objet.
- L'objet est en accès lecture seulement dans ses méthodes constantes.

Méthodes constantes

Intérêt : Protection supplémentaire

- Possibilité d'appeler des méthodes constantes sur des références constantes
- Impossibilité d'appeler une méthode non-constante sur une référence constante
- Le compilateur refuse de compiler.

Méthodes constantes : Exemple

- Dans la classe Personne, la méthode quiSuisJe est une bonne candidate comme méthode constante

```
void quiSuisJe() const;
```

```
void Personne::quiSuisJe() const {
  cout << "Je suis " << my_prenom << " " <<
    my_nom << endl;
  cout << "j'ai " << my_age << " ans" << endl;
}
```

- En revanche, aucun constructeur/destructeur ou opérateur d'affectation n'est une méthode constante

Méthodes constantes : Exercice

- Reprendre la classe ***Entier*** et désigner parmi les méthodes quelles sont celles (devant) être constantes
- Faites des essais depuis le main en instanciant des objets constants ou non constants
- Pourquoi la méthode surchargeant l'opérateur = ne peut pas être const?

Inclusions réciproques

- Il est interdit de faire de l'***inclusion réciproque*** dans les modules C++.
- Par exemple, si votre classe Point a besoin de la classe Vecteur et que la classe Vecteur a besoin de la classe Point alors on ne peut écrire :

fichier Point.h

```
#include "Vecteur.h"
...
class Point {
...
   Vecteur toVecteur( Point p );
...
};
```

fichier Vecteur.h

```
#include "Point.h"
...
class Vecteur {
...
   Point toPoint();
...
};
```

Inclusions réciproques : solution

- Si deux classe ont besoin de se connaître, il faut que l'une des classes n'utilise l'autre qu'à travers de ***pointeurs*** ou ***références*** dans l'***interface de la classe***.

fichier Point.h

```
#include "Vecteur.h"
...
class Point {
...
   Vecteur toVecteur( Point p );
...
};
```

fichier Vecteur.h

```
// La classe 'Point' existe quelque part.
class Point;
...
class Vecteur {
...
  // Retourne un pointeur sur un 'Point'.
  // (certainement alloue dyn.)
  Point* toPoint();
...
};
```

Inclusions réciproques : solution

- Si deux classe ont besoin de se connaître, il faut que l'une des classes n'utilise l'autre qu'à travers de ***pointeurs*** ou ***références*** dans l'***interface de la classe***.

fichier Vecteur.cc

```
#include "Vecteur.h"
// On inclue l'interface de 'Point' seulement dans le corps
// de la classe 'Vecteur'.
#include "Point.h"
...
Point* Vecteur::toPoint()
{
  // e.g., construit le Point au bout du vecteur place
  // a l'origine.
  return new Point( my_dx, my_dy );
}
```

Inclusions réciproques : solution

- De manière générale : si vous manipulez une classe X dans une interface au travers de références ou de pointeurs, il est bon de :
  - ***Déclarer*** la classe dans l'interface : class X;
  - Et ***d'inclure*** le module « X.h » dans le corps de la classe
- N'oubliez pas de vérifier votre conception objet lorsqu'un cas d'inclusion réciproque apparaît.

Inclusions réciproques : exercice

- Créer une classe B
- Créer une méthode affiche(A a) dans B
- Créer une classe A ayant pour attribut un pointeur vers B
- Créer dans A une méthode affiche() appelant la méthode affiche(A a) de B
- Créer le main pour tester tout ça

Inclusions réciproques : exercice

- Essayez de déclarer, dans B, un attribut de type A.
- Tout compile, mais il y a une erreur à l'exécution. Pourquoi?

Pointeur this

- Dans le corps d'une méthode d'une classe, vous disposez d'un moyen d'accéder à l'objet courant (ie soi même) : le pointeur ***this***.
- Si vous êtes dans le corps de la classe X, alors ***this*** est un pointeur (constant) de X.

Pointeur this : Exemple

```
void Personne::quiSuisJe() const {
  cout << "Je suis " << my_prenom << " " <<
    my_nom << endl;
  cout << "j'ai " << this->my_age << " ans" << endl;
}
```

- La première écriture est un raccourci offert par le C++ de la deuxième
- Le pointeur this est notamment utilisé dans l'opérateur d'affectation pour tester si l'objet reçu en paramètre n'est pas l'objet courant (ie sois même).
  - Ex : une instruction du genre : a = a;