Cours C++ : Initiation POO (deuxième partie)

© 2013 tv <tvaira@free.fr> - v.1.0

Sommaire

Notions avancées

Constructeur de copie

Le **constructeur de copie** est appelé dans :
– la création d'un objet à partir d'un autre objet pris comme modèle
– le passage en paramètre d'un objet par valeur à une fonction ou une méthode
– le retour d'une fonction ou une méthode renvoyant un objet

Remarque : toute autre duplication (au cours de la vie d'un objet) sera faite par l'opérateur d'affectation (=).

 © 2013 tv <tvaira@free.fr>

La forme habituelle d'un constructeur de copie est la suivante :

```
class T
{
   public:
      T(const T&);
};
```

Donc pour la classe `PileChar` :

```
PileChar::PileChar(const PileChar &p) : max(p.max), sommet(p.sommet)
{
   pile = new char[max]; // on alloue dynamiquement le tableau de caractère

   unsigned int i;

   // on recopie les éléments de la pile
   for (i = 0; i < sommet ; i++) pile[i] = p.pile[i];
   #ifdef DEBUG
   cout << "PileChar(const PileChar &p) : " << this << "\n";
   #endif
}
```

Deux situations où le constructeur de copie est nécessaire :

```
PileChar pile3a(pile2); // Appel du constructeur de copie pour instancier pile3a

PileChar pile3b = pile1b; // Appel du constructeur de copie pour instancier pile3b
```

Opérateur d'affectation

L'opérateur d'affectation (=) est un opérateur de copie d'un objet vers un autre. L'objet affecté est déjà créé sinon c'est le constructeur de copie qui sera appelé.

La forme habituelle d'opérateur d'affectation est la suivante :

```
class T
{
   public:
      T& operator=(const T&);
};
```

Cet opérateur renvoie une référence sur `T` afin de pouvoir l'utiliser avec d'autres affectations. En effet, l'opérateur d'affectation est associatif à droite a=b=c est évaluée comme a=(b=c). Ainsi, la valeur renvoyée par une affectation doit être à son tour modifiable.

La définition de l'opérateur = est la suivante :

```
PileChar& PileChar::operator = (const PileChar &p)
{
   // vérifions si on ne s'auto-copie pas !
   if (this != &p)
   {
      delete [] pile; // on libère l'ancienne pile
```

```
      max = p.max;
      sommet = p.sommet;
      pile = new char[max]; // on alloue une nouvelle pile
      unsigned int i;
      for (i = 0; i < sommet ; i++) pile[i] = p.pile[i]; // on recopie les éléments de la
          pile
   }
   #ifdef DEBUG
   cout << "operator= (const PileChar &p) : " << this << "\n";
   #endif
   return *this;
}
```

Ce qui permettra d'écrire :

```
PileChar pile4; // Appel du constructeur par défaut pour créer pile4

pile4 = pile3a; // Appel de l'opérateur d'affectation pour copier pile3a dans pile4
```

Surcharge d'opérateurs

La **surcharge d'opérateur** permet aux opérateurs du C++ d'avoir une signification spécifique quand ils sont appliqués à des types spécifiques. Parmi les nombreux exemples que l'on pourrait citer :
- `myString + yourString` pourrait servir à concaténer deux objets `string`
- `maDate++` pourrait servir à incrémenter un objet `Date`
- `a * b` pourrait servir à multiplier deux objets `Nombre`
- `e[i]` pourrait donner accès à un élément contenu dans un objet `Ensemble`

Les opérateurs C++ que l'on surcharge habituellement :
- Affectation, affectation avec opération (`=, +=, *=`, etc.) : **Méthode**
- Opérateur « fonction » `()` : **Méthode**
- Opérateur « indirection » `*` : **Méthode**
- Opérateur « crochets » `[]` : **Méthode**
- Incrémentation `++`, décrémentation `--` : **Méthode**
- Opérateur « flèche » et « flèche appel » `->` et `->*` : **Méthode**
- Opérateurs de décalage `<<` et `>>` : **Méthode**
- Opérateurs `new` et `delete` : **Méthode**
- Opérateurs de lecture et écriture sur flux `<<` et `>>` : **Fonction**
- Opérateurs dyadiques genre « arithmétique » (`+,-,/` etc) : **Fonction**

Les autres opérateurs ne peuvent pas soit être surchargés soit il est déconseillé de le faire.

La **première technique** pour surcharger les opérateurs consiste à les considérer comme des **méthodes** normales de la classe sur laquelle ils s'appliquent.

Le principe est le suivant :

```
A Op B se traduit par A.operatorOp(B)

t1 == t2; // équivalent à : t1.operator==(t2)
t1 += t2; // équivalent à : t1.operator+=(t2)
```

On surcharge les opérateurs `==`, `!=` et `+=` pour la classe `PileChar` :

```
class PileChar
{
   private:
      unsigned int max;
      unsigned int sommet;
      char *pile;

   public:
      ...
      bool operator == (const PileChar &p); // teste si deux piles sont identiques
      bool operator != (const PileChar &p); // teste si deux piles sont différentes
      PileChar& operator += (const PileChar &p); // empile une pile sur une autre
};

bool PileChar::operator == (const PileChar &p)
{
   if(max != p.max) return false;
   if(sommet != p.sommet) return false;
   unsigned int i;
   for (i = 0; i < sommet ; i++) if(pile[i] != p.pile[i]) return false;
   return true;
}

bool PileChar::operator != (const PileChar &p)
{
   // TODO
}

PileChar& PileChar::operator += (const PileChar &p)
{
   unsigned int i, j;

   // Reste-il assez de place pour empiler les caractères ?
   if((sommet + p.sommet) <= max)
   {
      for (i = sommet, j = 0; j < p.sommet ; i++, j++) pile[i] = p.pile[j]; // on recopie
          les éléments de la pile
      sommet += j; // on met à jour le nouveau sommet
   }
   else cerr << "Pile pleine !\n";

   return *this;
}
```

La **deuxième technique** utilise la surcharge d'opérateurs externes sous forme de **fonctions**. La définition de l'opérateur ne se fait plus dans la classe qui l'utilise, mais en dehors de celle-ci. Dans ce cas, tous les opérandes de l'opérateur devront être passés en paramètres.

Le principe est le suivant :

```
A Op B se traduit par operatorOp(A, B)

t1 + t2; // équivalent à : operator+(t1, t2)
```

Les opérateurs externes doivent être déclarés comme étant des **fonctions amies** (`friend`) de la classe sur laquelle ils travaillent, faute de quoi ils ne pourraient pas manipuler les données membres de leurs opérandes.

Exemple pour une classe `T` :

```
class T
{
   private:
      int x;

   public:
      friend T operator+(const T &a, const T &b);
};

T operator+(const T &a, const T &b)
{
   // solution n° 1 :
   T result = a;
   result.x += b.x;
   return result;

   // solution n° 2 : si l'opérateur += a été surchargé
   T result = a;
   return result += b;
}
```

```
T t1, t2, t3; // Des objets de type T

t3 = t1 + t2; // Appel de l'opérateur + puis de l'opérateur de copie (=)

t3 = t2 + t1; // idem car l'opérateur est symétrique
```

L'avantage de cette syntaxe est que l'opérateur est réellement symétrique, contrairement à ce qui se passe pour les opérateurs définis à l'intérieur de la classe.

Les attributs statiques

Un membre donnée déclaré avec l'attribut `static` est **partagé par tous les objets de la même classe**. Il existe même lorsque aucun objet de cette classe n'a été créé.

Un membre donnée statique doit être initialisé explicitement, à l'extérieur de la classe (même s'il est privé), en utilisant l'opérateur de résolution de portée (::) pour spécifier sa classe.

En général, son initialisation se fait dans le fichier `.cpp` de définition de la classe.

On va utiliser un membre statique `nbPoints` pour **compter le nombre d'objets `Point` créés à un instant donné**.

On **déclare** un membre donnée statique de la manière suivante :

```
class Point
{
   private:
      double x, y; // nous sommes des attributs de la classe Point
      static int nbPoints; // je suis un membre donnée statique

   ...
};
```

Point.h

Il faut maintenant `initialiser` ce membre statique et `compter/décompter` le nombre d'objets créés et détruits :

```
int Point::nbPoints = 0; // initialisation d'un membre statique

Point::Point() // Constructeur
{
   x = 0.;
   y = 0.;
   nbPoints++; // un objet Point de plus
}

...

Point::~Point() // Destructeur
{
   nbPoints--; // un objet Point de moins
}
```

Point.cpp

Tous les constructeurs de la classe `Point` doivent incrémenter le membre statique `nbPoints` !

Les fonctions membres statiques

Lorsqu'une fonction membre a **une action indépendante d'un quelconque objet de sa classe**, on peut la déclarer avec l'attribut `static`.

Dans ce cas, une telle fonction peut être appelée, sans mentionner d'objet particulier, en préfixant simplement son nom du nom de la classe concernée, suivi de l'opérateur de résolution de portée (::).

Les fonctions membre statiques :
- ne peuvent pas accéder aux attributs de la classe car il est possible qu'aucun objet de cette classe n'ait été créé.
- peuvent accéder aux membres données statiques car ceux-ci existent même lorsque aucun objet de cette classe n'a été créé.

On va utiliser une fonction membre statique `compte()` pour **connaître le nombre d'objets `Point` existants à un instant donné**.

On **déclare** une fonction membre statique de la manière suivante :

```
class Point
{
   private:
      double x, y; // nous sommes des attributs de la classe Point
      static int nbPoints; // je suis un membre donnée statique

   public:
      ...
      static int compte(); // je suis une méthode statique

};
```

Point.h

Il faut maintenant `définir` cette méthode statique :

```
// je retourne le nombre d'objets Point existants à un instant donné
int Point::compte()
{
   return nbPoints;
}
```

Point.cpp

La surcharge des opérateurs de flux « et »

Rappels : Un **flot** est un **canal recevant (flot d'« entrée ») ou fournissant (flot de « sortie ») de l'information**. Ce canal est associé à un périphérique ou à un fichier.

Un **flot d'entrée** est un objet de type `istream` tandis qu'un **flot de sortie** est un objet de type `ostream`.

Le flot `cout` est un flot de sortie prédéfini connecté à la sortie standard `stdout`. De même, le flot `cin` est un flot d'entrée prédéfini connecté à l'entrée standard `stdin`.

On surchargera les opérateurs de flux « et » pour une classe quelconque, sous forme de **fonctions amies**, en utilisant ces « canevas » :

```
ostream & operator << (ostream & sortie, const type_classe & objet1)
{
   // Envoi sur le flot sortie des membres de objet en utilisant
   // les possibilités classiques de << pour les types de base
   // c'est-à-dire des instructions de la forme :
   // sortie << ..... ;
   return sortie ;
}

istream & operator >> (istream & entree, type_de_base & objet)
{
   // Lecture des informations correspondant aux différents membres de objet
   // en utilisant les possibilités classiques de >> pour les types de base
   // c'est-à-dire des instructions de la forme :
   // entree >> ..... ;
   return entree ;
```

 © 2013 tv <tvaira@free.fr>

```
}
```

Si on implémente la surcharge de l'opérateurs de flux de sortie « pour qu'il affiche un objet `Point` de la manière suivante : `<x,y>`, on pourra alors écrire :

```
Point p0, p1(4, 0.0), p2(2.5, 2.5);

cout << "P0 = " << p0 << endl;
cout << "P1 = " << p1 << endl;
cout << "P2 = " << p2 << endl;
```

Ce qui donnera :

```
P0 = <0,0>
P1 = <4,0>
P2 = <2.5,2.5>
```

Pour obtenir cela, on écrira :

```
ostream & operator << (ostream & os, const Point & p)
{
   os << "<" << p.x << "," << p.y << ">";
   return os;
}
```

Idem pour la surcharge de l'opérateurs de flux d'entrée » pour qu'il réalise la saisie d'un objet `Point` de la manière suivante : `<x,y>`.

```
cout << "Entrez un point : ";
cin >> p0;

if (! cin)
{
   cout << "ERREUR de lecture !\n";
   return -1;
}

cout << "P0 = " << p0 << endl;
```

Ce qui donnera :

```
Entrez un point : <2,6>
P0 = <2,6>

Entrez un point : <s,k>
ERREUR de lecture !
```

L'amitié

Rappel : l'unité de protection en C++ est la classe. Ceci implique :
- Une fonction membre peut accéder à tous les membres de n'importe quel objet de sa classe ;
- Par contre, le principe d'encapsulation interdit à une fonction membre d'une classe d'accéder à des données privées d'une autre classe.

L'amitié en C++ permet à des fonctions (ou des méthodes) **d'accéder à des membres privées d'une autre classe**.

Les **fonctions amies** se déclarent en faisant précéder la déclaration classique de la fonction du mot clé **`friend`** à l'intérieur de la déclaration de la classe cible.

```
class A
{
    private:
        int a; // Un attribut privé

    public:
        A(int a=0) : a(a) {}
        friend void modifierA(int i); // Une fonction amie
};

void modifierA(int i)
{
    A unA;

    unA.a = i; // possible car je suis un ami de la classe A
}
```

Il est possible de déclarer amie une méthode d'une autre classe, en précisant son nom complet à l'aide de l'opérateur de résolution de portée.

Il est aussi possible que toutes les fonctions membres d'une classe soient amies d'une autre classe.

```
class A
{
    private:
        int a; // Un attribut privé

    public:
        A(int a=0) : a(a) {}
        friend class Amie; // Toutes les méthodes de la classe Amie sont mes amies
};

class Amie
{
    public:
        void afficherA(void)
        {
            A unA;

            cout << unA.a << endl; // produit : 0
        }
```

```
};

int main(void)
{
    Amie amie;
    amie.afficherA();
    return 0;
}
```

Dans ce cas, toutes les fonctions membres de la classe `Amie` deviennent des amies de la classe `A`.

On remarquera plusieurs choses importantes :
- **l'amitié n'est pas transitive** : les amis des amis ne sont pas des amis. Une classe A amie d'une classe B, elle-même amie d'une classe C, n'est pas amie de la classe C par défaut.
- **l'amitié n'est pas héritée** : mes amis ne sont pas les amis de mes enfants. Si une classe A est amie d'une classe B et que la classe C est une classe fille de la classe B, alors A n'est pas amie de la classe C par défaut.

L'utilisation des classes amies peut aussi traduire un défaut de conception (classes amies vs classes dérivées).

Les méthodes *inline*

C++ présente une **amélioration** des **macros** du langage C avec les **fonction `inline`** :
- Elles ont le comportement des fonctions (vérification des arguments et de la valeur de retour) ;
- Elles sont substituées dans le code après vérification.

Il existe deux techniques pour implémenter une fonction `inline` :

```
class Entier
{
    private:
      int a;
    public:
        // 1. lorsque le corps de la méthode est définie directement dans la déclaration
        int getEntier() const { return a; } // getEntier() est alors inline
        void setEntier(int i);
};

// 2. lorsqu'on définit une méthode, on ajoute au début de la définition le mot-clé inline
inline void Entier::setEntier(int i) // setEntier() est alors inline
{
    a = i;
}
```

Il est habituellement impératif que la définition de la fonction (la partie entre `{ ... }`) soit placée dans un fichier d'en-tête (extension `.h`). Si on la met dans un fichier d'implémentation `.cpp` dans le cas d'une compilation séparée, on aura "une erreur externe" (fonction non définie) au moment de l'édition de liens.

De manière générale, les fonctions `inline` (ou les macros) ont les caractéristiques suivantes :
- Avantage : plus rapide qu'une fonction (sauf si la taille du programme devient trop importante) ;
- Inconvénients : comme le code est généré pour chaque appel, le programme binaire produit est plus volumineux. D'autre part, cela complique la compilation séparée.