C++ Avancé (1° Partie)

Bjarne Stroustrup a développé C++ au cours des années 1980, alors qu'il travaillait dans le laboratoire de recherche Bell d'AT&T. Il s'agissait en l'occurrence d'améliorer le langage C. Il l'avait d'ailleurs nommé C with classes (« C avec des classes »).

Le langage C++ est normalisé par l'ISO. Sa première normalisation date de 1998 (ISO/CEI 14882:1998), sa dernière de 2003 (ISO/CEI 14882:2003). La normalisation de 1998 standardise la base du langage (Core Language) ainsi que la bibliothèque standard de C++ (C++ Standard Library).

Table des matières

Agrégation

- On distingue deux types d'agrégation :
  - Agrégation interne : l'objet agrégé est créé par l'objet agrégateur
  - Agrégation externe : l'objet agrégé a été créé extérieurement à l'objet agrégateur

- Il y a 3 possibilités de mise en oeuvre :
  - agrégation par valeur (appelée aussi composition)
  - agrégation par référence
  - agrégation par pointeur

➢ *Remarque : en règle générale, le créateur d'un objet est le responsable de sa destruction.*

Agrégation par valeur (Composition)


```
class ObjetGraphique
{
  public:
   ObjetGraphique(Point p, int couleur) : point(p)
   {
      this->couleur = couleur;
   }

  private:
   int    couleur;
   Point  point; // par valeur
};
```

Agrégation par référence


```
class ObjetGraphique
{
  public:
   ObjetGraphique(Point &p, int couleur): point(p)
   {
      this->couleur = couleur;
   }

  private:
   int    couleur;
   Point &point; // par référence
};
```

Agrégation par pointeur


```
class ObjetGraphique
{
  public:
   ObjetGraphique(Point *p, int couleur) : point(p)
   {  this->couleur = couleur; }
   ObjetGraphique(Point &p, int couleur) : point(&p)
   {  this->couleur = couleur; }

  private:
   int    couleur;
   Point *point; // par pointeur
};

// ObjetGraphique o1(&p1, c);
// ObjetGraphique o2(p2, c);
```

Objets Constants

- Les règles suivantes s'appliquent aux objets constants :
  - On déclare un objet constant avec le modificateur `const`
  - On ne peut appliquer que des méthodes constantes sur un objet constant
  - Un objet passé en paramètre sous forme de référence constante est considéré comme constant

- *Remarque : une méthode constante est tout simplement une méthode qui ne modifie aucun des attributs de l'objet*

Méthodes Constantes

- La qualification `const` d'une fonction membre fait partie de sa signature. Ainsi, on peut surcharger une fonction membre non constante par une fonction membre constante :

```
class Point
{
   private:
      int x, y;
   public:
      int X() const { return x; }
      int Y() const { return y; }
      int& X() { return x; }
      int& Y() { return y; }
};
```

- Il est conseillé de qualifier `const` toute fonction qui peut l'être.

Attributs Constants

- Une donnée membre d'une classe peut être qualifiée `const`.
- Il est alors obligatoire de l'initialiser lors de la construction d'un objet, et sa valeur ne pourra par la suite plus être modifiée.

```
class ObjetGraphique
{ public:
   ObjetGraphique(Point &p, int couleur, int num):point(p),
numero(num) { this->couleur = couleur; }

  private:
   int    couleur;
   Point &point;
   const int numero;
};
```

Passage de paramètres

- Il y a deux méthodes pour passer des paramètres dans une fonction ou une méthode :
  - le passage par valeur
  - le passage par variable (par référence ou par pointeur)

- Avantages et inconvénients des deux méthodes
  - Les passages par variables sont plus rapides et plus économes en mémoire que les passages par valeur. Il faut donc éviter les passages par valeur dans les cas d'appels récursifs de fonction ou avec des grandes structures de données.
  - Les passages par valeurs permettent d'éviter de détruire par mégarde les variables passées en paramètre. Si l'on veut se prévenir de la destruction accidentelle des paramètres passés par variable, il faut utiliser le mot clé `const`.

Références et pointeurs constants et volatiles

- L'utilisation des mots clés `const` et `volatile` avec les pointeurs et les références est un peu plus compliquée qu'avec les types simples. En effet, il est possible de déclarer des pointeurs sur des variables, des pointeurs constants sur des variables, des pointeurs sur des variables constantes et des pointeurs constants sur des variables constantes (idem avec les références). La position des mots clés `const` et volatile dans les déclarations des types complexes est donc extrêmement importante.

- *Remarque : lors de l'analyse de la déclaration d'un identificateur X, il faut toujours commencer par une phrase du type « X est un ... ». Pour trouver la suite de la phrase, il suffit de lire la déclaration en partant de l'identificateur et de suivre l'ordre imposé par les priorités des opérateurs. Cet ordre peut être modifié par la présence de parenthèses.*

Exercice I

- Décrivez les lignes de code suivantes :

```
1) const int *pi;

2) int const *pi;

3) int j; int * const pi = &j;

4) const int *pi[12];
```

Surcharge des opérateurs (1/x)

- Opérateurs du C++ ne pouvant être surchargés :
  - . : Sélection d'un membre
  - .* : Appel d'un pointeur de méthode membre
  - :: : Sélection de portée
  - :? : Opérateur ternaire
  - sizeof, typeid, static_cast, dynamic_cast, const_cast, reinterpret_cast

- Opérateurs C++ qu'il vaut mieux ne pas surcharger :
  - , : Évaluation séquentielle d'expressions
  - ! : Non logique
  - || && : Ou et Et logiques

Surcharge des opérateurs (2/x)

- 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

Surcharge des opérateurs internes

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

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

```
type &operator+=(const type &);
```

```
type &type::operator+=(const type &b)
{
    x += b.x;
    return *this;
}
```

Surcharge des opérateurs externes (1/2)

- La deuxième technique utilise la surcharge d'opérateurs externes. 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.

- La syntaxe est la suivante : `type operatorOp(opérandes)`

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

- 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.

Surcharge des opérateurs externes (2/2)


```
friend type operator+(const type &a, const type &b);

type operator+(const type &a, const type &b)
{
    type result = a;
    return result += b;
}
```

- *Remarque : on constatera que 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.*

Opérateurs de comparaison

- Les opérateurs de comparaison sont très simples à surcharger. La seule chose essentielle à retenir est qu'ils renvoient une valeur booléenne.

bool `type`::operator==(const type &) const;

Opérateurs d'incrémentation et de décrémentation (1/3)

- Les opérateurs d'incrémentation et de décrémentation ont la même notation mais représentent deux opérateurs en réalité. En effet, ils n'ont pas la même signification, selon qu'ils sont placés avant ou après leur opérande. Ne possédant pas de paramètres (ils ne travaillent que sur l'objet), il est donc impossible de les différencier par surcharge.
- La solution qui a été adoptée est de les différencier en donnant un paramètre fictif de type `int` à l'un d'entre eux.
  - opérateurs préfixés : ++ et -- ne prennent pas de paramètre et doivent renvoyer une référence sur l'objet lui-même
  - opérateurs suffixés : ++ et -- prennent un paramètre int fictif (que l'on n'utilisera pas) et peuvent se contenter de renvoyer la valeur de l'objet

Opérateurs d'incrémentation et de décrémentation (2/3)

- Sous forme de méthode :

```
type &type::operator++(void) // Opérateur préfixe :
{
   ++x ; // incrémente la variable
   return *this ; // et la retourne
}

// Opérateur suffixe : retourne la valeur et incrémente la variable
type type::operator++(int n)
{  // crée un objet temporaire,
   type tmp(x); // peut nuire gravement aux performances
   ++x;
   return tmp;
}
```

Opérateurs d'incrémentation et de décrémentation (3/3)

- Sous forme de fonctions amies :

```
type operator++(type &a) // Opérateur préfixe
{
   ++a.x;
   return a;
}

type operator++(type &a, int n) // Opérateur suffixe
{
   type tmp = a;
   ++a.x;
   return tmp;
}
```

Forme Canonique de Coplien

- Une classe T est dite sous forme canonique (ou forme normale ou forme standard) si elle présente les méthodes suivantes :

```
class T
{
   public:
      T (); // Constructeur par défaut
      T (const T&); // Constructeur de recopie
      ~T (); // Destructeur éventuellement virtuel
      T &operator=(const T&); // Operateur d'affectation
};
```

Constructeur par défaut

- Le constructeur par défaut est un constructeur sans argument ou avec des arguments qui possèdent une valeur par défaut.

- Il est nécessaire notamment dans :
  - la création de tableaux d'objet
  - la fabrication d'objets temporaires
  - l'agrégation par valeur des objets

- Attention : le compilateur peut fournir un constructeur par défaut automatique, qui varie selon les environnements de développement.

Constructeur de recopie (1/3)

- Le constructeur de recopie est appelé dans :
  - la création (déclaration) 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.*

Constructeur de recopie (2/3)

- La forme habituelle d'un constructeur de recopie est la suivante : `T::T(const T&)`

- L'objet modèle est passé :
  - en référence, ce qui assure de bonnes performances et empêche un bouclage infini
  - et la référence est constante, ce qui garantit que seules des méthodes constantes (ne pouvant pas modifier les attributs) seront appelables sur l'objet passé en argument

Constructeur de recopie (3/3)

- *Remarque : le compilateur fournit un constructeur de recopie automatique « par recopie bit à bit optimisée »*

- On aura besoin de définir son constructeur de recopie :
  - Agrégation par valeur : il faut recopier les objets agrégés par valeur en appelant leur constructeur par recopie
  - Agrégation par référence (ou pointeur) et allocation dynamique de mémoire : la copie optimisée ne générant que des copies de pointeurs, les différents objets utiliseraient les même blocs mémoire.

Destructeur

- Le destructeur sera nécessaire dès lors que la classe réalise de :
  - l'agrégation par pointeur
  - l'allocation dynamique de mémoire

Opérateur d'affectation

- Il est nécessaire de créer un opérateur de recopie :
  - agrégation
  - allocation dynamique de mémoire ;
- Dans tous les autres cas, le compilateur crée un opérateur d'affectation « par recopie bit à bit optimisée »
- La forme habituelle d'opérateur d'affectation est la suivante :

```
T& T::operator=(const T&)
```

- Cet opérateur renvoie une référence sur T (`return *this`) afin de pouvoir l'utiliser avec d'autres affectations.
- Rappel : 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

Exercice II

- Décrivez les lignes de code suivantes (mettant en jeu les classes T et U) :

```
1) T t;
2) U u;
3) T z(t);
4) T v();
5) T y=t;
6) z = t;
```

Gestion des exceptions (1/2)

- Les exceptions ont été rajoutées à la norme du C++ afin de faciliter la mise en oeuvre de code robuste.

Bloc de code protégé (unique)

```
try
{
  // Code susceptible de lever une exception
}
```

Gestionnaires d'exceptions (multiples)

```
catch (TypeException &identificateur)
{
  // Code de gestion d'une exception
}
```

n

- Découpage du traitement d'erreur en deux parties :
  - le déclenchement : instruction `throw`
  - le traitement : deux instructions inséparables `try` et `catch`

Gestion des exceptions (2/2)

- Une exception est levée ...
- Si l'instruction en faute n'est pas dans un bloc `try`, il y appel immédiat de la fonction `terminate`.
- Si l'instruction en faute est incluse dans un bloc `try`, le programme saute directement vers les gestionnaires d'exception qu'il examine séquentiellement dans l'ordre du code source :
  - Si l'un des gestionnaires correspond au type de l'exception, il est exécuté, et, s'il ne provoque pas lui même d'interruption ou ne met fin à l'exécution du programme, l'exécution se poursuit à la première ligne de code suivant l'ensemble des gestionnaires d'interruption. En aucun cas il n'est possible de poursuivre l'exécution à la suite de la ligne de code fautive.
  - Si aucun gestionnaire ne correspond au type de l'exception, celle-ci est propagée au niveau supérieur de traitement des exceptions (cas de blocs `try` imbriqués) jusqu'à arriver au programme principal qui lui appellera `terminate`.

(Re)lancer une exception


```
#include <stdexcept> // pour std::range_error
double inverse(double x)
{   if(x==0.0)
      // lance une exception
      throw range_error("Division par zero !\n") ;
    else return 1/x;
}

try
{ ... }
catch (range_error &e)
{
   // traitement local
   throw; // relance l'exception
}
```

Déclarer ses exceptions (1/2)

- Selon la norme, les exceptions peuvent être de n'importe quel type (y compris un simple entier). Toutefois, il est utile de les définir en tant que classes. Pour cela, on dérive la classe fournit en standard `std::exception` et on surchargera au minimum la méthode `what()`.

```
class ErreurX: public exception
{   public:
      ErreurX() throw() {}
      ~ErreurX() throw() {}
      const char *what(void) const throw()
      { // on peut aussi utiliser un string en privé
         return "Exception sur ...";
      }
};
```

Déclarer ses exceptions (2/2)


```
try
{
   // bloc de code protégé
}
catch (ErreurX &e)
{
   cerr << "Erreur : " << e.what() << endl;
}
catch (exception &e)
{
   cerr << “Exception inconnue : “ << e.what() << endl;
   cerr << “Fin du programme” << endl ; // ou pas
   exit(1);
}
```

Les spécificateurs d'exception

- Un spécificateur d'exception renseigne l'utilisateur sur le type des exceptions que peut renvoyer une méthode.

```
class A
{
   private:
      int x;
   public:
      A(int x = 0) throw (ErreurX);
      ~Ratio();
};
```

- Mais, le spécificateur d'exception interdit aux autres méthodes (ou fonctions) appelées d'invoquer des exceptions non prévues. Hors, ce point est difficile à vérifier lors de la compilation. Aussi, les spécificateurs d'exception doivent ils être réservés au code maîtrisé totalement et plus spécifiquement aux méthodes pour lesquelles on est en mesure de prévoir le déroulement complet.

Bibliographie

- Cours d'Éric REMY : http://pluton.up.univ-mrs.fr/eremy/Ens/Info1.C++/CM/C++.html

- Cours et tutoriels C++ : http://cpp.developpez.com/cours/