POO2

Victor Lezaud

22 janvier 2018

Sommaire

1 Amitié

1.1 Principe

La relation d'amitié permet de contourner l'encapsulation en permettant l'accès à des attributs privés à l'extérieur de la classe. Cela permet d'améliorer les performances en évitant d'utiliser les services de la classe, mais il ne faut pas en abuser car cela viole le principe d'encapsulation. On peut déclarer une relation d'amitié avec le mot `friend` entre une classe et :

- une fonction ordinaire
- une méthode d'une autre classe
- une classe complète.

1.2 Mise en oeuvre

```
// Interface de la classe <C1> (fichier C1.h)
#define C1_H
class C2;
class C1
{
   public :
      // amitié de la fonction non-membre operator <<
      friend std::ostream & operator << (std::ostream & out, const C1 & o);

      // amitié de la méthode m de la classe C2
      friend const char * C2::m(const C1 & o);

      // amitié d'une classe C3
      friend class C3;

   private :
      int x;
}
```

1.3 Limite de la Relation d'Amitié

La relation d'amitié n'est présente que si elle est déclaré explicitement. Il n'y a :

- ni symétrie
- ni transitivité
- ni héritage

2 La surcharge

2.1 Principe

2.1.1 But

La surcharge permet de définir plusieurs fonctions/méthodes avec le même nom. Il suffit que les différentes versions n'aient pas la même signature.

2.1.2 Signature d'une fonction

La signature d'une fonction est composée de son nom et du nombre, l'ordre et le type de ses paramètres formels. Le type de retour ne fait pas partie de la signature

2.1.3 Paramètres par défaut

```
int f(int n=0, int d=1);
```

La fonction f ci-dessus a 3 signatures différentes :
- f(int,int)
- f(int)
- f()

On ne peut donc pas définir une nouvelle fonction f sans paramètres car cela créerait une ambiguïté

2.2 Subtilités de la surcharge

```
void f (int x){}
void f (const int x){} // ERREUR : le const ne permet
                       //  pas de faire la différence
void f (int* x){}
void f (const int* x){} // CORRECTE : le const permet
                       // de faire la diff avec les pt
void f (int & x){}
void f (const int & x){} // CORRECTE : comme les pointeurs
```

2.3 Choix de la fonction/méthode par le compilateur

2.3.1 Règle 1

On appelle prioritairement la fonction/méthode dont les paramètres correspondent parfaitement (liste, ordre et type)

2.3.2 Règle 2

Si aucune fonction/méthode ne satisfait la règle 1, on utilise des conversions de types. Cette règle est très pratique mais source de nombreux problèmes

2.4 Surcharge des Opérateurs

2.4.1 Généralités

Permet de remplacer une écriture fonctionnelle par une représentation algébrique. L'arité, la priorité et l'associativité sont les mêmes que pour les opérateurs d'origine. Cette technique rend le code plus lisible mais peut-être moins performante (dû aux nombreuses copies d'objet). Il faut garder l'esprit de l'opérateur. Les opérateurs suivants ne peuvent être surchargés :
- : :
- .
- .*
- ? :

2.4.2 Opérateur interne : fonction memebre

Principe

```
class c1{
...
   type operator Op (parametres);
...
}
```

Le premier opérande est l'objet appelant la fonction. S'il y a d'autres opérandes, ils sont passés en paramètre de la fonction. Le type de retour est très souvent le type de l'appelant, on utilise pour cela le pointeur this. L'opérateur interne est efficace pour les opérateurs modifiant l'appelant.

Exemple : +=

```
class type{
...
   type operator += (type op2); // déclaration de la surcharge
...
}
// les deux lignes ci-dessous sont équivalentes :
x.operator += (y);
x += y;
```

2.4.3 Opérateur externe : fonction ordinaire

```
type operator Op (parametres);
```

Les paramètres sont les opérandes. Retour par valeur ou par référence d'un objet. On déclare souvent les opérateurs amis des classes pour faciliter l'accès aux attributs.

Exemple : +

```
type operator + (type op1, type op2); // déclaration de la surcharge

// les deux lignes ci-dessous sont équivalentes :
operator + (x,y);
x + y;
```

2.4.4 L'affectation

L'operator = est comme le constructeur de copie : le compilateur en crée un par défaut en cas d'absence mais cette version par défaut ne réalise q'une copie en surface. Il faut donc surcharger cette opérateur dés lors que l'on utilise un pointeur dans la classe.

2.4.5 Surcharge des opérateurs d'E/S

Permet d'insérer ou d'extraire des données selon des règles standard du langage C++. On retourne par référence le flux manipulé pour permettre les enchaînements

```
istream & operator >> ( istream & flux, const T & valeur );
//istream désigne la classe flux en entrée
ostream & operator << ( ostream & flux, const T & valeur );
//ostream désigne la classe flux en sortie
```

2.4.6 Opérateur ++ et −

Ces opérateurs définissent à la fois la post-incrémentation et la pré-incrémentation (x++,++x). On ajoute donc un paramètre fictif int (non-utilisé) pour la post-incrémentation. Le retour diffère pour les deux :

- pré : référence de l'objet
- post : valeur de l'objet

2.5 Généricité

2.5.1 Généralités

Un template est une entité qui peut définir un ensemble de classe ou de fonctions. La programmation générique permet de réduire la taille du code et de factoriser le travail.

2.5.2 Fonction générique

```
template <parametres_template>
type nomFonction (parametres_fonction);
```

Exemple :

```
template <typename T >
T Minimum ( const T & x, const T & y )
{
        return x < y ? x : y;
}
double prix(14.99);
cout << Minimum(prix,7.43) << endl;
// Déduction automatique de la fonction générique
cout << Minimum<int>(prix,5) << endl;
// Il faut préciser car il y a ambiguïté
cout << Minimum(int(prix),5) << endl;
// Ici on a supprimé l'ambiguïté
```

Les fonctions génériques doivent être déclarées et définies dans une interface (.h). Les opérations effectuées doivent exister pour tous les types passés en paramètres.

Spécialisation : On peut réaliser des spécialisations des fonctions génériques, cela revient à définir un comportement particulier de la fonction pour une certaine valeur de paramètre template. Dans ce cas la spécialisation doit être écrite après la fonction générique.

```
template < >
Point Minimum < Point > (const Point & p1, const Point & p2)
{
        ...
}
```

2.5.3 Classe Générique

```
template < typename T >
class Point
{
        public :
                void Deplacer(const Point < T > & delta);
                Point ( const Point < T > & unPoint);
                Point ( T abs = T ( ), T ord = T ( ) );
                ~Point ( );

        protected:
                T x;
                T y;
}

template < typename T >
void Point < T >::Deplacer (const Point < T > & delta)
{
        ...
}
template < typename T >
Point < T >::Point (T abs, T ord)
{
        ...
}
...
```

L'ensemble des méthodes et des fonctions liées à une classe template doivent être définies dans le fichier .h où la classe est définie afin d'être sur que le compilateur crée les spécialisations de ces fonctions/méthodes pour toutes les utilisations de la classe.

2.5.4 Classe Générique et Paramétrage

Il est aussi possible de passer des paramètres template autre que des types. On peut par exemple donner un int pour paramétrer la taille d'un tableau statique. Attention du code sera généré pour chaque valeur distincte donnée aux paramètres templates.

2.5.5 Forme Canonique d'une Classe

La forme canonique d'une classe permet d'utiliser celle-ci de façon similaire à un objet de type de base. En cas de non-définition de ces méthodes, le compilateur fournit une version minimaliste et souvent insuffisante en présence d'attributs dynamiques. Elle contient :

- la surcharge de l'affection
- le constructeur par défaut
- le constructeur de copie
- le destructeur

3 Librairies d'entrées sorties en C++

3.1 Généralités

Les services d'entrées/sorties sont basés sur les flux. Ils sont définis grâce à un hiérarchie de classes génériques. Le type le plus utilisé étant le char il possède un instanciation spécifique (celle qu'on utilise tout le temps). Il en existe aussi une pour le wchar_t.

3.2 Décomposition des classes de la librairie

3.2.1 Abstraction vs Implémentation

Il y a deux catégories de classes :

- Les classes d'abstractions
  - définissent une interface capable de fonctionner avec n'importe quel type de flux
  - ne nécessite pas de connaître la localisation exacte de la donnée (fichier, mémoire, socket, ...) qui est lue ou écrite
- Les classes d'implémentations
  - Ces classes héritent des classes d'abstraction et définissent une implémentation spécifique pour un type précis de source de données
  - Disponible dans la librairie standard
    - Une implémentation pour les streams basés sur des fichiers
    - Une implémentation pour les streams basés sur des tempons mémoires (memory buffer)

3.2.2 Bas niveau vs Haut niveau

On peut décomposer les classes de la librairie en 2 groupes :

- Pour les opérations de bas niveau
  - Classes streams buffers
  - Elles agissent sur des caractères sans fournir aucune possibilités de formatage
  - Elles sont rarement utilisées directement
- Pour les opérations de haut niveau
  - Classes stream
  - Elles fournissent de nombreuses possibilités de formatage
  - Elles s'appuient sur les classes stream buffers

3.3 Entrées / Sorties basées sur les streams

Il y a deux étapes dans une opération d'entrée / sortie en C++ :
- Le formatage des données (à la charge d'une classe stream)
- La communication vers/depuis un périphérique externe
  - A la charge d'une classe stream buffer, représentation interne su stream dans lequel on lit/écrit.
  - Toutes les classes gérant un flux possèdent un objet de la classe streambuf.
  - Récupération possible du pointeur sur le stream buffer courant du flux à l'aide de la méthode ios : :rdbuf()
  - Modification possible de ce pointeur (redirection)

3.4 Etat d'un flux

3.4.1 Généralités

Chaque flux possède un vecteur de bits définissant ses indicateurs d'erreur :
- goodbit : activé, si l'état est correct
- eofbit : activé, si la fin de fichier a été atteinte sur le flux d'entrée
- failbit : activé, si la dernière opération d'entrée/sortie a échoué
- badbit : activé, si la dernière opération d'entrée/sortie est invalide

Positionné automatiquement par les opérations d'entrée/sortie.

3.4.2 Manipulation

On peut manipuler ces états :
- Globalement grâce à 3 méthodes publiques de la classe basic_ios : rdstate, setstate et clear.
- Individuellement grâce à 4 méthodes publiques de la classe basic_ios : good, eof, fail et bad

3.4.3 Exemple de manipulation

Et bit à bit :

```
ifstream fic;
fic.rdstate() & ifstream::failbit // permet de tester le failbit
```

operator bool renvoie true s'il n'y a pas d'erreur sur le flux et qu'il est prêt.

```
while(fic.get(carLu)) // utilisation dans une boucle
        cout << carLu;

if ( fic )
{        // lecture du fichier
}
else
{
        cerr << "Erreur d'ouverture du fichier"<<endl;
}
```

operator ! renvoit true si failbit ou badbit est positionné.

3.5 Formatage avec le type fmtflags

Les indicateurs de format sont définis avec un type ios_base : :fmtflags. C'est un champ de bits pour représenter les différents indicateurs. On utilise des constantes nommées (de champ de bit) publique de la classe ios_base. On les manipule avec des fonctions membres :

- fmtflags flags ( ) const ;
  - Renvoie le formatage actuellement actif sur le flux
- fmtflags flags (fmtflags fmtfl)
  - Positionne le formatage sur le flux
  - Les indicateurs non définis par fmtfl sont réinitialisés
  - Renvoie le formatage actif avant la modification sur le flux
- void unsetf (fmtflags mask) ;
  - Réinitialise les indicateurs de format du flux définis par le paramètre mask
  - Définition possible de mask avec une combinaison "ou" d'indicateurs de format
- fmtflags setf (fmtflags fmtfl) ;
  - Positionne le formatage sur le flux en laissant les autres inchangés
  - Renvoie le formatage actif avant la modification (permet une restauration)
- fmtflags setf (fmtflags fmtfl, fmtflags mask) ;
  - Positionne les indicateurs de format dont les bits sont activés à la fois dans fmtfl et dans mask et réinitialise les indicateurs de format dont les bits sont définis dans mask mais pas dans fmtfl
  - Renvoie le formatage actif avent la modification sur le flux

3.6 Formatage avec fonction membre

- width : détermine le nombre minimum de caractères à écrire
- precision : détermine la précision des flottants
- fill : valeur du caractère de remplissage

3.7 Manipulateurs

3.7.1 Généralités

Les manipulateurs dont des fonctions globales issues de de <iomanip>. On les utilise avec les opérateurs « et » sur des objets streams iostream. Ils modifient le comportement, les propriétés et les caractéristiques de formatage des streams.

3.7.2 Syntaxe d'utilisation

```
// Appel de fonction
nomManipulateur ( nomFlux );
endl ( cout );

// Utilisation de la surcharge de << ou >>
```

```
nomFlux << nomManipulateur;
cout << endl;
```

3.7.3 Ecriture de son propre manipulateur

```
const char ROUGE [ ] = { 033,'[','3','1','m'};
ostream & rouge ( ostream & os )
{
        os.write ( ROUGE, sizeof ( ROUGE ) );
        return os;
}
```

Un manipulateur est une fonction passée en paramètre aux opérateurs d'insertion et d'extraction dans un flux. Il est appelé par ces opérateurs. Pour passer des paramètres à un manipulateur on crée une classe correspondant au manipulateur. On stocke les valeurs passés au constructeur dans des attributs privés. On définit une sucharge de l'opérateur « ou » amie de la classe.

```
class Width
{
        public:
                explicit Width( int x ) : largeur ( x ) { }
                inline friend ostream & operator <<
                      ( ostream & os, const Width & manip)
                {
                        os.width(largeur);
                        return os;
                }
        private:
                int largeur;
}
```

4 Standard Template Library (STL)

4.1 Généralités

La STL est une bibliothèque C++, normalisé par l'ISO. Elle est composé de nombreux composants efficaces et réutilisables. Elle est disponible dans divers environnements de développement.

4.2 Contenu

La STL contient :
— Un ensemble de classes conteneurs
— Une abstraction des pointeurs : itérateurs
— Des algorithmes génériques : insertion, suppression, tri, recherche...
— Une classe string
— Un mécanisme de bas-niveau pour l'allocation et la libération de mémoire
— Une généralisation des fonctions

4.3 Itérateur

4.3.1 Caractéristiques

- Objet utilisé pour parcourir les éléments d'un conteneur
- Objet utilisé par les algorithmes
- S'appuie sur un ensemble d'opérateurs, avec au minimum :
  - L'incrémentation
  - Le déréférencement

4.3.2 6 catégories d'itérateurs

- InputIterator et OutputIterator
  - Les plus limités : opérations d'entrée et de sortie séquentielles dans un seul sens
- ForwardIterator
  - Mêmes opérations qu'un itérateur input et s'il n'est pas constant, il possède les mêmes possibilités qu'un itérateur output
  - Déplacement du début vers la fin (dans un seul sens)
  - Tous les conteneurs supportent cet itérateur
- BiderectionalIterator
  - Mêmes fonctionnalités qu'un itérateur forward mais avec des possibilités de déplacement dans les 2 sens
- RandomAccessIterator
  - Fonctionnalités similaires à un pointeur qui est d'ailleurs un itérateur de cette catégorie
  - Accès direct à un élément possible sans itérer à travers tous les éléments.
- ContiguousIterator (C++17)

4.3.3 Détails

Un itérateur est associé à une séquence d'éléments, il désigne une position dans cette séquence. On désigne le début et la fin de la séquence par des itérateurs (obtenu avec les méthodes begin(), end(), rbegin() et rend() des conteneurs)

4.3.4 Itérateur d'insertion

L'itérateur d'insertion est une sorte de output iterator. Il pertmet d'insérer un élément dans un conteneur :

- A une position donnée : insert_iterator
  - Si ii est un insert_iterator alors *ii = x réalise l'insertion de l'élément x dans le conteneur c à la position de l'itérateur
  - Cela correspond à c.insert(p,x)
- En début : front_insert_iterator
- En fin : back_insert_iterator

Les itérateurs d'insertion sont des adaptateurs d'itérateur. Les itérateurs inserter, front_inserter et back_inserter construisent automatiquement un itérateur d'insertion à partir de ses arguments (inférence de type)

4.3.5 Itérateur inversé

Les itérateurs inversés sont des adaptateurs d'itérateur. Ils permettent de parcourir un conteneur en sens inverse.

4.4 Conteneurs

4.4.1 Caractéristiques

— Objet support qui stocke une collection d'autres objets (ses éléments)
— Implémentation sous forme de modèles de classe (template)
— Gestion complète de l'espace mémoire alloué aux éléments
— Manipulation possible des éléments à partir des services (méthodes)

4.4.2 Différents conteneurs

— Séquence
— Adaptateur
— Associatif
— Associatif non ordonné

4.5 Algorithme