C++ Avancé (3° Partie)

Bjarne Stroustrup a développé C++ au cours des années 1980, alors qu'il travaillait dans le laboratoire de recherche Bell d'AT&T.

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).

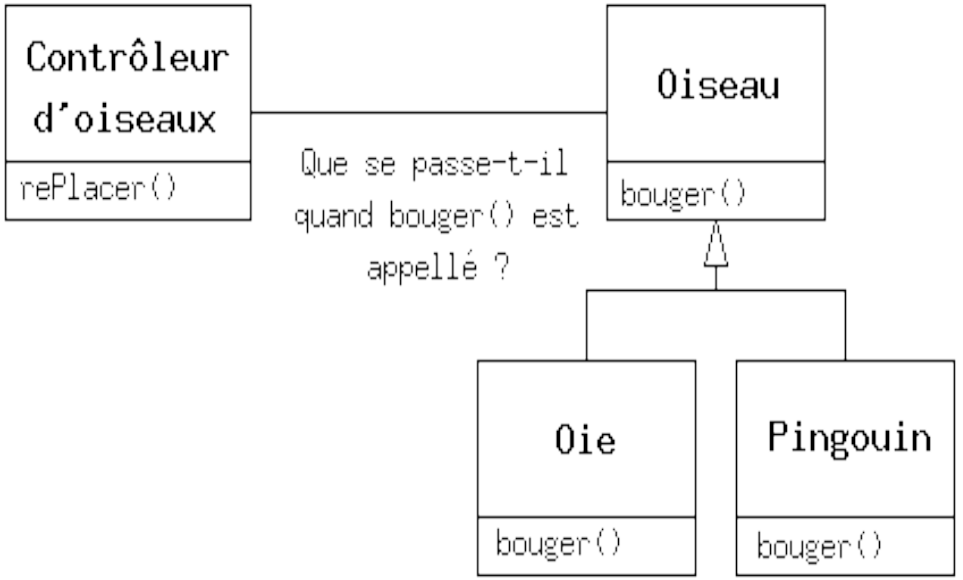
La relation « est un » est exprimée par l'héritage, et la relation « a un » est exprimée par la composition.

Table des matières

Polymorphisme
Transtypage
STL

Comportement Polymorphe (1/4)

- Par exemple, dans le diagramme suivant, l'objet Contrôleur d'oiseaux travaille seulement avec des objets Oiseaux génériques, et ne sait pas de quel type ils sont. C'est pratique du point de vue de Contrôleur d'oiseaux, car il n'a pas besoin d'écrire du code spécifique pour déterminer le type exact d'Oiseau avec lequel il travaille, ou le comportement de cet Oiseau.
- Comment se fait-il donc que, lorsque bouger() est appelé tout en ignorant le type spécifique de l'Oiseau, on obtienne le bon comportement (une Oie court, vole ou nage, et un Pingouin court ou nage) ?



Comportement Polymorphe (2/4)

- La réponse constitue l'astuce fondamentale de la programmation orientée objet : le compilateur ne peut faire un appel de fonction au sens traditionnel du terme. Un appel de fonction généré par un compilateur non orienté objet crée ce qu'on appelle une association prédéfinie : le compilateur génère un appel à un nom de fonction spécifique, et l'éditeur de liens résout cet appel à l'adresse absolue du code à exécuter.

- En POO, le programme ne peut déterminer l'adresse du code avant la phase d'exécution, un autre mécanisme est donc nécessaire quand un message est envoyé à un objet générique.

Comportement Polymorphe (3/4)

- Pour résoudre ce problème, les langages orientés objet utilisent le concept d' association tardive. Quand un objet reçoit un message, le code appelé n'est pas déterminé avant l'exécution. Le compilateur s'assure que la fonction existe et vérifie le type des arguments et de la valeur de retour, mais il ne sait pas exactement quel est le code à exécuter.
- Pour créer une association tardive, le compilateur C++ insère une portion spéciale de code en lieu et place de l'appel absolu. Ce code calcule l'adresse du corps de la fonction, en utilisant des informations stockées dans l'objet. Ainsi, chaque objet peut se comporter différemment suivant le contenu de cette portion spéciale de code. Quand un objet reçoit un message, l'objet sait quoi faire de ce message.

Comportement Polymorphe (4/4)

- On déclare qu'on veut une fonction qui ait la flexibilité des propriétés de l'association tardive en utilisant le mot-clé `virtual`. On n'a pas besoin de comprendre les mécanismes de virtualpour l'utiliser, mais sans lui on ne peut pas faire de la programmation orientée objet en C++. En C++, on doit se souvenir d'ajouter le mot-clé `virtual` parce que, par défaut, les fonctions membre ne sont pas liées dynamiquement. Les fonctions virtuelles permettent d'exprimer des différences de comportement entre des classes de la même famille.
- Ces différences sont ce qui engendre un **comportement polymorphe**.

Transtypage

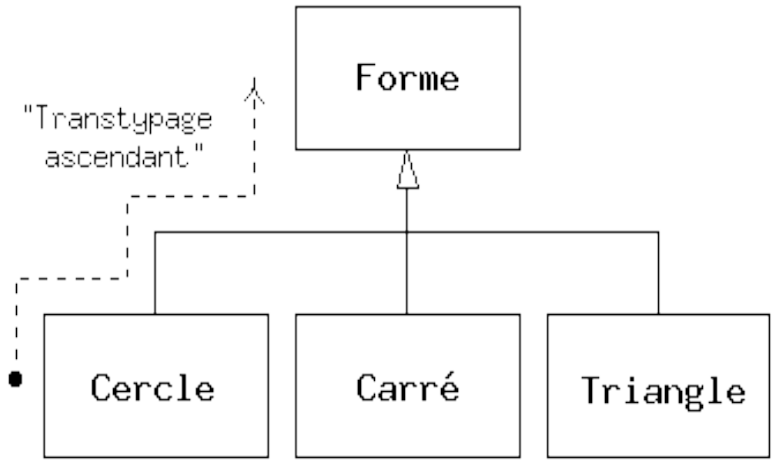
- Le transtypage (*cast* ou conversion de type) en C++

- Nouvelle syntaxe de l'opérateur traditionnel :
  - En C : `(nouveau type)(expression à transtyper)`
  - En C++ : `type(expression à transtyper)`

- Deux situations en cas d'héritage :
  - transtypage « ascendant » (*upcast*) : changer un type vers son type de base
  - transtypage « descendant » (*downcast*) : conversion d'un pointeur sur un objet d'une classe générale vers un objet d'une classe spécialisée.

Transtypage «ascendant»


```
class Forme {};
class Cercle : public Forme {};
class Triangle : public Forme {} ;
void faireQuelqueChose(Forme &f) {
  f.dessiner(); }
Cercle c; Triangle t;
faireQuelqueChose(c); faireQuelqueChose(t);
```



- Un Cercle est ici passé à une fonction qui attend une Forme. Comme un Cercle **est une** Forme, il peut être traité comme tel par faireQuelqueChose().
- Traiter un type dérivé comme s'il était son type de base est appelé transtypage ascendant, surtypage ou généralisation (*upcasting*).

Nouveaux opérateurs de transtypage

- static_cast : Opérateur de transtypage à tout faire. Ne permet pas de supprimer le caractère const ou volatile.
- const_cast : Opérateur spécialisé et limité au traitement des caractères const et volatile
- dynamic_cast : Opérateur spécialisé et limité au traitement des downcast.
- reinterpret_cast : Opérateur spécialisé dans le traitement des conversions de pointeurs peu portables (permet de réinterpréter les données d'un type en un autre type. Aucune vérification de la validité de cette opération n'est faite).

- La syntaxe est la suivante :

```
op_cast<expression type>(expression à transtyper)
```

  - Où op prend l'une des valeurs (static, const, dynamic ou reinterpret)

L'opérateur static_cast

- C'est l'opérateur de transtypage à tout faire qui remplace dans la plupart des cas l'opérateur hérité du C.
- Toutefois il est limité dans les cas suivants :
  - Il ne peut convertir un type constant en type non constant
  - Il ne peut pas effectuer de promotion (*downcast*)
- Exemple :

```
int i;
double d;
i = static_cast<int>(d);
```

L'opérateur const_cast (1/2)

- Il est spécialisé dans l'ajout ou le retrait des modificateurs `const` et `volatile`.
- En revanche, il ne peut pas changer le type de données de l'expression. Ainsi, les seules modifications qu'il peut faire sont les suivantes :
  - X -> const X
  - const X -> X
  - X -> volatile X
  - volatile X -> X
  - const X -> volatile X
  - volatile X -> const X

L'opérateur const_cast (2/2)

- Exemple :

```
Toto t1(1);
const Toto t2(2);
Toto *t11 = &t1;
const Toto *t22 = &t2;
const Toto *t33;

//t11 = &t2; // erreur: invalid conversion from ‘const
Toto*’ to ‘Toto*’
//t11 = static_cast<Toto *>(&t2); // erreur: static_cast
from type ‘const Toto*’ to type ‘Toto*’

t11 = const_cast<Toto *>(&t2); // Ok !
t33 = const_cast<const Toto *>(&t2); // Ok !
```

L'opérateur dynamic_cast (1/4)

- Le transtypage dynamique permet de convertir une expression en un pointeur ou une référence d'une classe, ou un pointeur sur void. Il est réalisé à l'aide de l'opérateur dynamic_cast. Cet opérateur impose des restrictions lors des transtypages afin de garantir une plus grande fiabilité :
  - il effectue une vérification de la validité du transtypage ;
  - il n'est pas possible d'éliminer les qualifications de constance (pour cela, il faut utiliser l'opérateur const_cast).

- Il ne peut pas travailler sur les types de base du langage, sauf void *.

L'opérateur dynamic_cast (2/4)

- Les downcast posent un problème particulier car leur vérification n'est possible qu'à l'exécution. Aussi, contrairement à static_cast et const_cast qui ne sont que des informations à l'adresse du compilateur, dynamic_cast génère du code de vérification.

- La syntaxe de l'opérateur dynamic_cast est donnée ci-dessous :

```
dynamic_cast<type>(expression)
```

L'opérateur dynamic_cast (3/4)

- Exemple :

```
Ligne *l;
Cercle *c;
Forme **formes = new Forme*[2];
Forme *forme = new Forme();

formes[0] = new Ligne(1,1,0,2);
formes[1] = new Cercle(2,2,0,3);

forme = formes[0];
forme->afficher() ;

//l = forme; // erreur: invalid conversion from ‘Forme*’ to ‘Ligne*’
l = dynamic_cast<Ligne *>(forme); // ok
cout << l << endl;
l->afficher();
```

L'opérateur dynamic_cast (4/4)

- En revanche, s'il n'est pas du bon type, dynamic_cast n'effectue pas le transtypage. Si le type cible est un pointeur, le pointeur nul est renvoyé. Exemple :

```
forme = formes[0];
c = dynamic_cast<Cercle *>(forme); // Aie !
cout << c << endl;
c->afficher(); // Erreur de segmentation
```

- Si en revanche l'expression caractérise un objet ou une référence d'objet, une exception de type bad_cast est lancée. Exemple :

```
try {
forme = formes[0];
Cercle &c1 = dynamic_cast<Cercle &>(*forme); // Aie !
cout << c << endl; c->afficher();
}
catch(bad_cast &e) {
cerr << "Echec transtypage : " << e.what() << endl; }
```

Transtypage et Polymorphisme (1/2)


```
class Animal {};
class Chien : public Animal {};
class Chat : public Animal {};

int main() {
   Animal* a = new Chat; // Transtypage ascendant : ok !

   // Essaye de le transtyper en Chat* :
   Chat* c2 = dynamic_cast<Chat *>(a); // Transtypage
descendant : erreur: cannot dynamic_cast ‘a’ (of type
‘class Animal*’) to type ‘class Chat*’ (source type is not
polymorphic)
}
```

Transtypage et Polymorphisme (2/2)


```
class Animal { public: virtual ~Animal() {} };
class Chien : public Animal {};
class Chat : public Animal {};

int main() {
   Animal* a = new Chat; // Transtypage ascendant
   cout << "a = " << (long)a << endl;

   // Essaye de le transtyper en Chat* :
   Chat* c2 = dynamic_cast<Chat *>(a); // Transtypage
descendant
   cout << "c2 = " << (long)c2 << endl;
}
```

```
a = 135970824
c2 = 135970824
```

Le besoin de conteneurs

- Situation : vous ne savez pas de combien d'objets vous allez avoir besoin lorsque vous écrivez le programme. Par exemple, dans un système de contrôle du trafic aérien vous ne voulez pas limiter le nombre d'avions que votre système peut gérer. Le programme planterait juste parce que vous avez dépassé un certain nombre.
- Mauvaise approche : créer un énorme tableau global d'objets (surcoût de temps d'appel du constructeur et du destructeur peut ralentir les choses significativement)
- Bonne approche : quand vous avez besoin d'un objet, le créer avec new, et placer son pointeur dans un conteneur. Plus tard, le "repêcher" et faire quelque chose avec. Ainsi, on ne crée que les objets dont on a absolument besoin.

Rôle des itérateurs

- Un itérateur est un objet qui parcourt un conteneur d'autres objets et en sélectionne un à la fois, sans fournir d'accès direct à l'implémentation de ce conteneur. Les itérateurs fournissent une manière standardisée d'accéder aux éléments, qu'un conteneur fournisse ou non un moyen d'accéder directement aux éléments.
- De bien des façons, un itérateur peut être considéré comme un "pointeur futé".

Librairie STL (Standard Template Library)

- La STL est une librairie qui est constituée principalement par des classes containeurs (vecteurs, listes ...), ainsi que des fonctionnalités pour parcourir (iterateur) leur contenu et des algorithmes pour travailler sur leur contenu(tri ...).

- Techniquement parlant le terme "STL" n'est plus significatif car ces classes on été pleinement intégrés à la librairie standard (std) ainsi que d'autres classes telles que iostream, aussi, en parlant de ces classes, on utilise souvent le terme std. Toutefois beaucoup de gens se réfèrent à la STL comme si elle était une chose séparée, donc vous entendrez le terme STL à la place de std.

Les conteneurs

- Les conteneurs sont une abstraction de stockage. Il s'agit de structures permettant d'organiser un ensemble de données. Les classes de conteneurs définies par la STL admettent donc toutes un paramètre template, qui, lors de l'instanciation d'un conteneur, sert à indiquer le type des objets que contiendra cette instance de conteneur.

Conteneurs séquentiels	vector	accès direct aux éléments, ajout et suppression efficaces en fin, mais linéaire en temps ailleurs
	deque (double en queue)	accès direct, ajout et suppression efficace en début ou en fin, linéaire ailleur
	list	liste (doublement) chaînée accès séquentiel, ajout et suppression en temps constant n'importe où dans la liste
Conteneurs associatifs	set (ensemble)	éléments maintenus ordonnés : appartenance, insertion et suppressions efficaces
	multiset	set avec possibilité de répétition
	map(carte, table)	accès aux valeurs via des clés ajout et suppressions efficaces
	multimap	map autorisant des clés dupliquées.

La Classe list

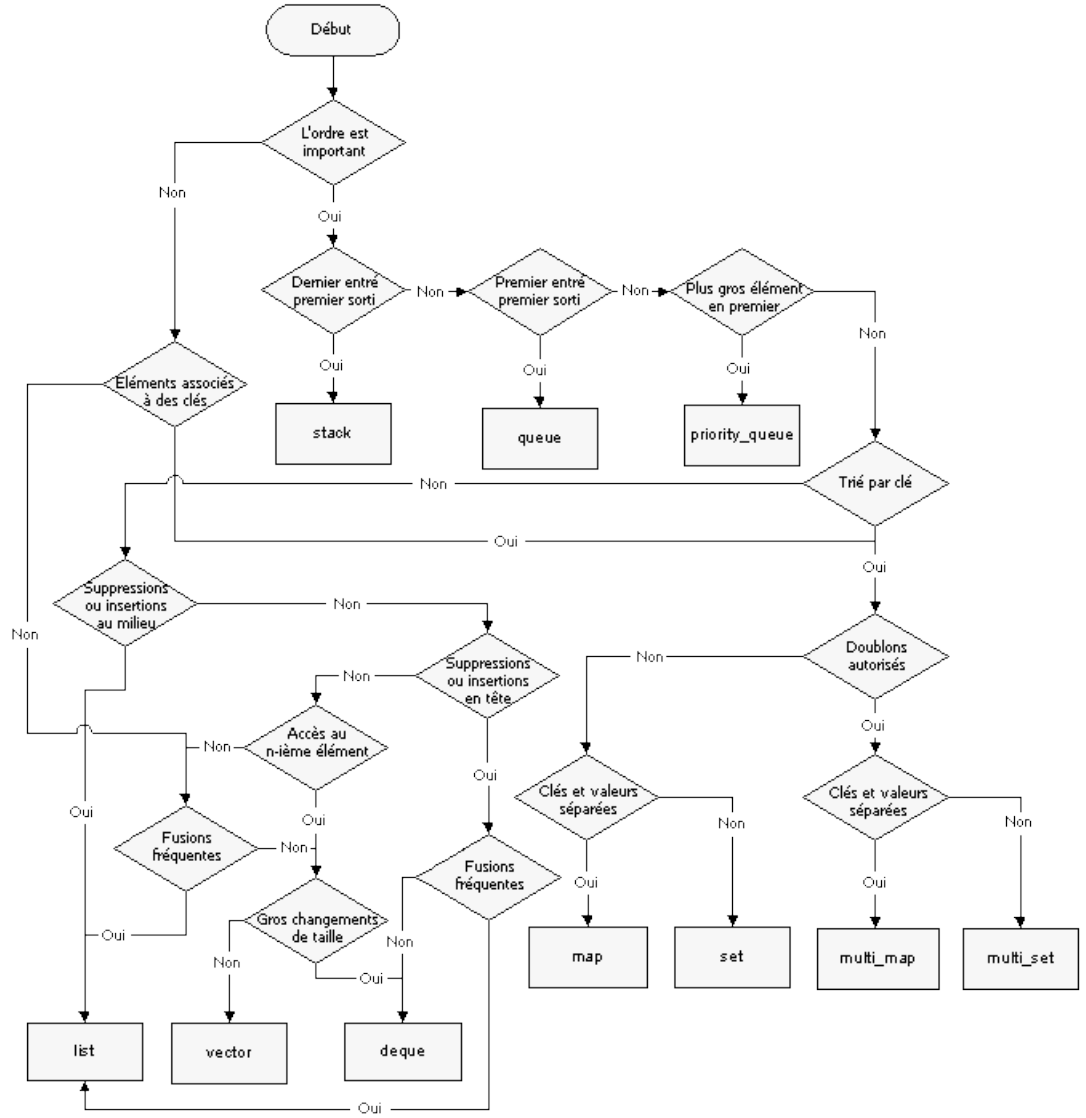
- La classe list est un conteneur ou chaque élément de la liste a son propre segment de mémoire et pointe son prédécesseur et son successeur (liste doublement chaînée). Exemple :

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

int main () {
  list<int> mylist; int myint;

  cout << "Please enter some integers (enter 0 to end):\n";
  do { cin >> myint;
    mylist.push_back(myint);
  } while (myint);

  cout << "mylist stores " << (int) mylist.size() << "
numbers.\n";
}
```

Itérateur


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

int main () {
  int myints[] = {75,23,65,42,13};
  list<int> mylist (myints,myints+5);

  list<int>::iterator it;

  cout << "mylist contains:";
  for ( it=mylist.begin() ; it != mylist.end(); it++ )
    cout << " " << *it;

  cout << endl;
}
```

Choix d'un conteneur



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/
- C++ Reference : http://www.cplusplus.com/reference/
- Standard Template Library Programmer's Guide : http://www.sgi.com/tech/stl/