Centrale-Lyon - Programmation Orientée Objet - Cours Info 2A (4/4)

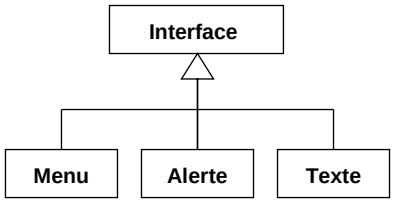
Programmation Orientée Objet en C++ (3ème partie)

- Polymorphisme
- Généricité

Version 2.2 – 8/11/2005

Polymorphisme

- Un objet polymorphe est un objet susceptible de prendre plusieurs formes **pendant l'exécution**
- Exemple :
  - les classes Menu, Alerte et Texte sont dérivées de la classe Interface ; ce sont des sortes d'interfaces
  - un objet instance de la classe Interface peut, durant l'exécution, prendre la forme d'un Menu, d'une Alerte ou d'un Texte
  - c-à-d qu'à une variable du type Interface on peut affecter toute **instance d'une classe dérivée d'Interface** (il y a changement de type automatique)



- Problème des objets polymorphes : sélection des méthodes correspondant à la forme de l'objet au moment de l'appel

Polymorphisme en C++

- En C++ :
  - le polymorphisme ne peut s'appliquer qu'à des **objets dynamiques** instances de classes constituant une hiérarchie
  - Les objets dynamiques sont donc manipulés à travers des pointeurs qui sont transtypés lors des changements de formes
- Exemple :
  - *ptr est un objet polymorphe ayant ici la forme d'un Menu

```
class Interface
{  … };

class Menu : public Interface
{  … };

class Alerte : public Interface
{  … };

class Texte : public Interface
{  … };

int main (void)
{
   Interface * ptr;
   Menu * ptrMenu;
   ptrMenu = new Menu (…);
   ptr = ptrMenu;
}
```

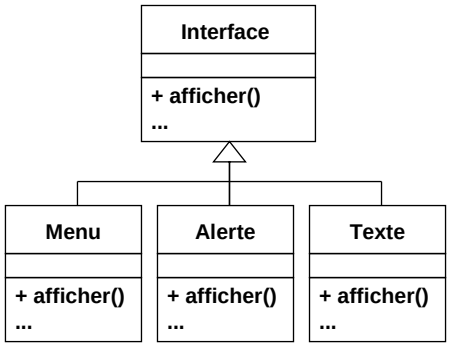
Cours Info 2A (4/4) C++ (3ème partie) 3

Problème de sélection des méthodes (1/2)

- Prenons le cas de la méthode *afficher* :
  - elle est surchargée dans chaque classe dérivée
  - dans le code suivant :

```
Interface * ptr
ptr = new Menu (…);
ptr->afficher ();
```

c'est la méthode *afficher* de Interface qui est appelée et non *afficher* de Menu comme on le voudrait !



- Explication :
  - le mauvais choix est dû à la liaison statique effectuée à la compilation (early binding), ptr est lié définitivement à la classe Interface même si à l'exécution ptr pointe sur un objet Menu

Cours Info 2A (4/4) C++ (3ème partie) 4

Problème de sélection des méthodes (2/2)

- Solution :
  - il faut mettre en œuvre une liaison dynamique entre un objet et une méthode à l'éxécution (late binding)
  - dans ce cas la méthode appelée sera bien celle de la classe dérivée et non celle de la classe de base
- Mise en œuvre :
  - les méthodes pour lesquelles on veut réaliser une liaison dynamique doivent être déclarées "virtuelles" dans la classe de base dont dérivent les différentes formes d'un objet polymorphe
- Déclaration des fonctions virtuelles en C++
  - Pour déclarer un fonction comme virtuelle, on fait précéder sa définition par le mot-clé ***virtual*** au niveau de la **classe de base**
  - les fonctions virtuelles doivent avoir le même prototype dans la classe de base et dans les classes dérivées
  - un destructeur peut être virtuel, mais pas un constructeur

Cours Info 2A (4/4) C++ (3ème partie) 5

Exemple de méthode virtuelle

- Ici la méthode appelée est bien *affiche* de la classe Menu

```
class Interface
{
  …
public:
    virtual void affiche () {  }
    …
 };

class Menu : public Interface
{
  …
public:
    Menu (...);
    ~Menu ();
    void affiche ();
};
```

```
class Alerte : public Interface
{   …  };

class Texte : public Interface
{   …  };

int main (void)
{
    Interface * ptr;
    Menu * ptrMenu;
    ptrMenu = new Menu (…);
    ptr = ptrMenu;
    …
    ptr -> affiche ();  // affiche de Menu !
    ...
}
```

Cours Info 2A (4/4) C++ (3ème partie) 6

Méthodes virtuelles pures, classe abstraites

- Méthodes virtuelles pures :
  - les méthodes virtuelles vues ci-avant ont une implémentation dans la classe de base (éventuellement sans code : { })
  - la méthode est **virtuelle pure** si elle n'a pas d'implémentation définie dans la classe de base
  - elle doit être redéfinie dans les classes héritières

```
class Base
{
   …
public:
   virtual void f (int x) = 0;
};
class Derivee : public Base
{
   …
public:
   void f (int x)  { … }   // f est redéfinie
};
```

- Classes abstraites :
  - Une classe est abstraite si elle contient au moins une méthode virtuelle pure
  - Une classe abstraite ne peut pas avoir d'instance

Cours Info 2A (4/4) C++ (3ème partie) 7

Fonctions génériques (1/3)

- Problème posé :
  - le programmeur doit souvent effectuer un même traitement sur des données de types divers (tri d'entiers, tri de réels, etc…)
- Plusieurs solutions :
  - écriture de plusieurs fonctions différentes
    - ➔ noms différents, problème de maintenance
  - écriture d'une fonction avec des arguments pointeurs génériques
    - ➔ pas de contrôle de type, le programmeur doit faire les conversions de pointeur
  - hiérarchie de classes dont les feuilles correspondent aux différents types de données
    - ➔ maintenance des diverses implémentations de la fonction
  - écriture d'une macro générique avec le type de données en paramètre
    - ➔ pas de contrôle du compilateur

Cours Info 2A (4/4) C++ (3ème partie) 8

Fonctions génériques (2/3)

- Solution au problème précédent :
  - écriture d'une fonction générique (modèle de fonction)
  - instanciation automatique pour divers types de données
- Définition d'une fonction générique :
  - les mots-clés template et class introduisent une fonction générique
  - les types susceptibles de remplacer T doivent posséder l'opérateur de comparaison '>' pour que le texte de *maxi* ait un sens
  - on peut avoir plusieurs paramètres de généricité ; chacun doit apparaître au moins une fois dans les paramètres de la fonction

```
template<class T> T maxi (T x, T y)
{
   return ( ( x > y ) ? x : y ) ;
} ;
```

```
template<class T, class X>
int f (T a, X b)
{  … };
```

Cours Info 2A (4/4) C++ (3ème partie) 9

Fonctions génériques (3/3)

- Instanciation et utilisation d'une fonction générique :
  - l'instanciation est automatique :
    - → avec des classes définies par l'utilisateur
    - → avec les types de base (pourvu que l'opérateur '>' soit défini pour ce type)
  - le programmeur peut fournir une ou plusieurs instances explicites particulières de la fonction
    - → cas des chaînes de caractères
  - lors d'un appel de fonction :
    - → recherche parmi les instances explicites
    - → si aucune ne convient, utilisation d'une fonction générique

```
class Truc
{  … } ;

int main (void)
{
   Truc a, b, c ;
   int x, y, z ;
   …
   a = maxi (b,c) ;
   x = maxi (y,z) ;
} ;
```

```
char * maxi (char * x, char * y)
{  return (strcmp (x,y)>0 ? x : y ); }
```

Cours Info 2A (4/4) C++ (3ème partie) 10

Classes génériques (1/4)

- Problème posé :
  - on a souvent à réaliser des structures de données s'appuyant sur des types variés (pile d'entiers, pile de réels, etc…)
- Solutions :
  - il est possible de mettre en œuvre des solutions diverses dans l'esprit de celles proposées pour les fonctions avec des problèmes similaires :
    - ➔ écriture de plusieurs classes
    - ➔ écriture d'un classe manipulant des pointeurs génériques
    - ➔ écriture de macros avec le type en paramètre
  - La "bonne" solution utilise la notion de classe générique (modèle de classe) et l'on met en œuvre diverses instanciations de celle-ci

Cours Info 2A (4/4) C++ (3ème partie) 11

Classes génériques (2/4)

- Définition d'une classe générique :
  - les mots-clés template et class introduisent une classe générique paramétrée par le type T

```
template<class T> class Pile
{
   T * lapile;
   int taille;
   int top;
public:
   Pile (int t)     { lapile = new T[taille=t]; top = -1; }
   ~Pile ()         { delete[] lapile; }
   void empile (const T & e)    { lapile [++top] = e; }
   T & depile()  { return lapile [top--]; }
};
```

- Instanciation et utilisation :
  - déclaration de 2 piles
  - NB : l'utilisation de typedef allège les notations :
    - ➔ typedef pile<int> pile_int;

```
int main (void)
{
   Pile<int> p1 (10);
   Pile<double> p2 (5);
   …
   p1.empile (20);
   y = p2.depile();
}
```

Cours Info 2A (4/4) C++ (3ème partie) 12

Classes génériques (3/4)

- Méthodes non-inline
  - il faut utiliser la syntaxe (lourde) suivante :

```
template<class T> void Pile<T>::empile (const T & e)
{
   lapile [++top] = e;
}
```

- Une classe générique peut être paramétrée par des types et par des variables
  - création d'une pile de 20 entiers

```
template<class T, int t> class Pile
{
   T * lapile;
   int taille;
   int top;
public:
   Pile ()      { lapile = new T[taille=t]; top = -1; }
   ~Pile ()     { delete[] lapile; }
   void empile (const T & e)    { lapile [++top] = e; }
   T & depile()  { return lapile [top--]; }
} ;
…
Pile<int,20> mapile;
```


Classes génériques (4/4)

- Instances spécifiques
  - l'instanciation d'une classe générique peut être impossible pour certains types
  - Le programmeur peut donc introduire des instances spécifiques de la classe
  - les instances spécifiques sont d'abord prises en compte avant d'effectuer des instanciations automatiques

```
class Pile<char>
{
   …
   // traitements spécifiques
   …
} ;
```

- Classe générique et fonctions amies
  - une fonction friend qui fait référence à un paramètre de généricité doit elle aussi être générique
  - une fonction friend générique n'est amie que des instances de la classe générique qui ont les **mêmes paramètres** de généricité
- Classe générique et attribut de classe
  - les attributs de classe (déclarés *static*) sont communs à tous les objets instances d'une instance particulière d'une classe générique