Programmation C++
Deuxième Partie

Michael Mrissa

Département Informatique
UFR Sciences et Techniques
Université de Pau et des Pays de l'Adour

Note

- Ce support est une évolution de celui de Nicolas Belloir
- Il est inspiré des supports de
  - Jean-Michel Bruel (IUT Blagnac)
  - Alain Dancel (`http://www.multimania.com/dancel/cplusplus/cours/cours_cpp.html`)
  - Marc Daniel (Ecole Supérieure d'Ingénieurs de Luminy)

*****

La forme canonique de Coplien

La forme canonique de Coplien

- Concerne les classes non triviales
- Fournit un cadre à respecter

Définition

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

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

Constructeur par défaut

- constructeur sans argument ou dont chacun des arguments possède une valeur par défaut
- Utilisé (par exemple) lorsque que l'on crée des tableaux d'objets (impossible de passer un constructeur à l'opérateur `new []`)
- Si aucun constructeur n'est spécifié, alors le compilateur essaie de fournir un constructeur par défaut.

Exemple

```
// Exemple erroné
class T {
    public:
        T (int i);          // Constructeur
};
int main(){
    T tab [3]               // Erreur car il faudrait un constructeur
                            //par defaut
}
// Exemple corrigé
class T {
    public:
        T (int i=3);          // Constructeur
};
int main(){
    T tab [3]               // Correct! Tous les objets sont
                            // construits avec la valeur 3
}
```

Constructeur par recopie

- permet la **création** d'un objet **à partir d'un autre objet** pris comme modèle.
  - Hors création, on utilisera l'opérateur d'affectation.
- Est utilisé pour passer un objet **par valeur** à une méthode. Garantit de ne pas modifier l'objet original
- Est utilisé pour retourner un objet comme retour d'une méthode. L'objet créé est placé dans la pile pour manipulation par le reste du programme.

Exemple

```
// prototype
    T::T (const T&);
```

Utilisation du constructeur par recopie

- Si le constructeur par recopie n'est pas spécifié, le compilateur en crée un.
- Cas où la création est obligatoire :
  - si présence d'objets agréés par valeur. Il est nécessaire de les recopier en appelant leur constructeur par recopie
  - Si présence d'objets agréés par référence. Recopie par défaut ne recopie que les pointeurs !

Opérateur d'affectation

- permet la **duplication** d'un objet **à partir d'un autre objet** pris comme modèle
- Obéit aux même règles que le constructeur par recopie
- Attention si l'objet recevant l'affectation n'est pas vierge
  - Utilisation du destructeur avant la recopie
-

Exemple

```
// prototype
    T& T::operator=(const T&);
```

*****

Quelques types de fonctions particulières

Les fonctions amies en C++

- **Entorse** aux concepts de la POO
- Traditionnellement, la partie private d'une classe est réservée aux fonctions membres de la classe
- Les amis (**friend**) permettent de **casser le mécanisme d'encapsulation**
- les Friends peuvent être
  - des fonctions indépendantes
  - des fonctions membres d'une autre classe
  - des fonctions amies de plusieurs classes
  - des classes amies

Attention

L'abus de friends est dangereux

Accès privilégié à une classe

- Exemple de la multiplication par une matrice (beaucoup d'appel) :

Exemple

```
T2 f(const T1& m, const T2& v) {
    ...
    m.get_val()...v.get_val()...
}
```

- En rendant `f()` "**amie**" de `T1` et `T2`, on obtient :

Exemple

```
T2 f(const T1& m, const T2& v) {
    ...m.val...v.val...
```

Méthodes constantes

- indique qu'une méthode n'est pas intrusive (pas de modif. de `*this`)
- obligatoire pour porter sur un objet constant

Exemple

```
class Forme {
    ...
    double get_x () const;
}; void foo(const Forme& f) {
    f.get_x(); // OK
}
```

*****

Surcharge des opérateurs

La surcharge d'opérateurs

- Un constat
  - Il est plus intuitif et plus clair d'additionner deux matrices en surchargeant l'opérateur d'addition et en écrivant :
    - `result = m0 + m1;`
  - Que d'écrire :
    - `matrice_add(result, m0, m1);`

Idée maître

Etendre les opérateurs de base pour qu'ils puissent s'appliquer aux objets

Règles d'utilisation

- La plupart des opérateurs sont surchargeables.
  - Les opérateurs ne peuvent être surchargés pour les types de base (int . . .)
- Il faut veiller à **respecter l'esprit de l'opérateur**.
- Lorsque l'on surcharge un opérateur, il n'est **pas possible de** :
  - changer sa priorité
  - changer son associativité
  - changer sa pluralité (unaire, binaire, ternaire)
  - créer de nouveaux opérateurs

Implémentation

- Quand l'opérateur `+` (par exemple) est appelé, le compilateur génère un appel à la fonction `operator+`. Ainsi, l'instruction `a = b + c;` est **équivalente** aux instructions :
  - `a = operator+(b, c); // fonction globale`
  - `a = b.operator+(c); // fonction membre`

Exemple pour l'opérateur d'affectation

Exemple

```
class Point {
    float x;
    float y;
};

Point & operator=(const Point & p){
    x = p.x;
    y = p.y;
    return *this;
}
```

Deux implémentations possibles

- par une fonction **membre**
  - implicitement, l'opérateur a accès à un paramètre de plus (`this`)
  - pas de symétrie naturelle (non commutatif)
  - si le membre de gauche est une classe : impossible !
  - non `static`
- par une fonction **non membre**
  - doit être amie pour accéder simplement aux membres
  - symétrie naturelle
  - pas d'accès à `this`

Comment choisir ?

- Cela dépend
  - des besoins de conception de la classe
    - fonctions membres $\Leftrightarrow$ accès possible à `this`
  - des besoins des utilisateurs
    - fonctions non membres $\Leftrightarrow$ symétrie naturelle
- Une règle :
  - $=$, (), [], $\rightarrow$: nécessairement fonctions membres
  - Opérateurs unaires : fonctions membres
  - Opérateurs d'affectation : fonctions membres
  - Opérateurs binaires : fonctions non membres amies

Il est préférable de

- Redéfinir l'opérateur d'**affectation**
- Redéfinir les opérateurs d'**égalité** et de **différence**
- Redéfinir les opérateurs << et >>
- Redéfinir l'opérateur d'**indexation** peut être très pratique

Exemple

```
class Point {
    float x;
    float y;

    friend ostream & operator << (ostream & flot, const Point &
    p){
            return (flot << "Les coordonnées sont : " << p.x << "
            et " << p.y;
    }
};
}
```

*****

L'héritage

L'héritage simple

- L'héritage permet de **créer** une nouvelle classe **à partir d'une classe déjà existante**
- La nouvelle classe hérite de tous les membres, qui ne sont pas privés, de la classe de base

Exemple

```
// classe parente :
class A {
    // membres
};

// classe dérivée :
class B : public A {
    // membres ajoutés ou redéfinis
};
```

Formes d'héritage

- Principe de substitution de Liskov
  - "Les méthodes qui utilisent des objets d'une classe doivent pouvoir utiliser des objets dérivés de cette classe sans même le savoir."
  - classe mère = interface implémentée par des classes filles
- Réutilisation du code
  - Ne pas confondre avec la composition

Formes d'héritage C++

- Public / Protected / Private
  - class B : public A {...} ;
  - class B : protected A {...} ;
  - class B : private A {...} ;

	Type héritage	Statut dans la classe de base	Statut dans la classe dérivée
Mode de dérivation	public	public	public
		protected	protected
		private	inaccessible
	protected	public	protected
		protected	protected
		private	inaccessible
	private	public	private
		protected	private
		private	inaccessible

TABLE – Tableau résumé de l'accès aux membres

L'héritage multiple

- En C++, il est possible d'utiliser l'héritage multiple

```
class A {
  public:
    void fa();
  protected:
    int _x;
};
```

```
class B {
  public:
    void fb();
  protected:
    int _x;
};
```

```
class C: public B,
         public A {
  public:
    void fc();
};
```

```
void C::fc() {
 int i;
 fa();
 i = A::_x + B::_x;
}
```

L'héritage multiple : gestion de conflits

- Que se passe-t-il en cas de **noms identiques** ?
- Utiliser l'**opérateur de portée**

Exemple

```
class A {
    public: int a;}
class B {
    public: int a;}
class C : public A, public B {
    public:
        int a;
        void f() {
            a = A::a + B::a;
...};
```

Construction et destruction

- Le constructeur de la classe mère est appelé avant toute autre chose
- Possibilité de spécifier des paramètres au constructeur de la classe mère

Exemple

```
class Personne {
  char * nom, *prenom;
  ...
};
class Etudiant : public Personne {
   double moyenne;
 public:
   Etudiant(double m, const char * name,
       const char * adress = 0) : Personne(name, adress), moyenne(m) {}
   ...
};
```

- le destructeur de la mère est appelé en dernier

Que ce passe t'il (cherchez l'erreur) ?

Exemple

```
class Cercle : public Forme {...};

...

Cercle c;

Forme f = c;

Forme * ptr1 = &c;

Cercle * ptr2 = &f;
```

*****

Méthodes virtuelles et classes abstraites

Méthodes virtuelles

- Comportement désiré :

Exemple

```
Forme f; Cercle c; Forme * ptr;

cout << f.surface(); // appel de Forme::surface()

cout << c.surface(); // appel de Cercle::surface()

cout << ptr->surface(); // appel de [classe pointée]::surface()
```

- Problème !
  - appel systématique de `Forme::surface()`
  - car le pointeur pointe sur une forme
- Solution
  - utiliser une méthode "virtuelle"
  - ajouter le mot-clé `virtual` devant la méthode

Méthodes virtuelles (suite)

- Possibilité de redéfinir une méthode de la classe mère

Exemple

```
Forme *tab[]={&c,&t};

for (...) tab[i]->surface();

  // pb : appel de Forme::surface()
  // solution : virtual double surface();
```

- Remarques :
  - un constructeur ne peut pas être virtuel
  - si une méthode est virtuelle alors le destructeur (éventuel) de la classe doit l'être aussi
  - méthodes virtuelles pures (cf. classes abstraites)

Classes abstraites

- Classe avec au moins une méthode virtuelle pure
- Notation : virtual type nomFonction (params) const = 0 ;
- Pas d'instance possible, car il manque au moins une implémentation d'une fonction
- Se rapproche d'une interface (interface = classe abstraite avec aucune implémentation)

Exemple

```
class Forme { ...
  virtual double surface () const = 0;
};
```