Université de Reims Champagne-Ardenne
UFR Sciences Exactes et Naturelles
Licence Informatique
INFO0402: Programmation Orientée Objet

TD N°3

Exercice 1: Différences entre membre et héritage

On a vu que le C++permettait la définition suivante :

```
struct A1 {
  int a1;
  A1(int x) : a1(x) {}
  int get() const { return a1; }
};
```

```
struct A2 {
  int a2;
  A2(int x) : a2(x) {}
  int get() const { return a2; }
};
```

```
struct B : A1 {
  A2  x;
};
```

Remarque : le mot-clé `struct` autorise aussi l'emploi `public`/`protected`/`private`. Le mot-clé `class` force leurs utilisations fixant `private` comme défaut.

1. Expliquer la différence sémantique entre un membre et un héritage.
2. Expliquer la différence entre un membre et un héritage en terme d'occupation mémoire.
3. Que se passe-t-il en terme d'alignement mémoire pour les membres et les classes héritées.
4. Pourquoi l'héritage a tendance à produire plus de trou dans les structures que les membres ?

Solution:

1. Membre = a un, Héritage = est un.
   Cela se traduit par l'accès suivant :
   - pour un héritage, la classe mère héritée est fusionnée avec ses membres et méthodes à la classe fille.
     impossible de séparer la classe de la classe héritée.
   - pour une déclaration comme membre, l'objet membre reste encapsulé dans sa boîte.

   ```
   B   b;
   int u,v;
   // accès à un champs/méthode hérité
   u = b.a1;
   v = b.get();   // retourne le a1 hérité
   // accès à un membre
   u = b.x.a2;
   v = b.x.get(); // retourne le a2 dans A2
   ```

2. comme les champs, les classes héritées sont placées dans l'ordre dans lequel elles sont héritées (pas dans la norme, mais constitue le moyen trivial d'obtenir les propriétés recherchées par l'héritage).
3. placement dans l'ordre dans lequel elle est déclarée/héritée avec alignement pour chaque membre/classe héritée sur l'adresse de son BIT le plus grand.
4. Si on peut choisir l'ordre dans lequel on organise les champs, on choisit rarement l'ordre dans lequel on construit une hiérarchie (découle de la logique de la conception, et non d'un choix d'organisation des champs).

Exercice 2: Héritage, constructeur et destructeur

On veut comprendre comment utiliser les différents constructeurs et destructeur dans le cas d'une chaine d'héritage. Afin de limiter au maximum l'accès aux champs, tous les champs seront privés, et les héritages publics.

Soit `A1` et `A2` deux classes exactement identiques (pour l'exemple), qui contiennent toutes les deux un entier, définissent toutes les deux, un constructeur par défaut (entier=0), un constructeur avec un entier (donne la valeur de l'entier), un constructeur par copie, une assignation par copie et un destructeur.

1. définir une classe A1 (la classe A2 sera identique en remplaçant A1 par A2).
2. soit la classe B qui hérite de A1 et possède un champs a2 de type A2. Définir cette classe sans aucune méthode.
3. indiquer comment est défini le constructeur par défaut de B, et le résultat de son exécution.
4. écrire le code du constructeur par défaut équivalent.
5. indiquer comment est défini par défaut le constructeur par copie, et le résultat de son exécution.
6. écrire le code du constructeur par copie équivalent.
7. indiquer comment est défini par défaut l'assignation par copie.
8. écrire le code de l'assignation par copie équivalent.
9. indiquer comment est défini le destructeur par défaut.
10. écrire le code du destructeur par défaut équivalent.
11. indiquer comment sont définis le constructeur et l'assignation par déplacement par défaut.

Solution:

1. voir le code suivant :

```
class A1 {
private: int a1;
public:
  A1() : a1(0) { cout << "A1::A1()" << endl; }
  A1(int v) : a1(v) { cout << "A1::A1(int)" << endl;; }
  A1(const A1& v) : a1(v.a1) { cout << "A1::A1(A1&)}
  ~A1() { cout << "A1::~A1()" << endl; }
  A1 &operator=(const A1& v) {
    cout << "A1::=(A1&)" << endl;
    if (&v != this) a1 = v.a1;
    return *this;
  }
  int get() const { return a1; }
  void set(int x) { a1 = x; }
};
```

2. voir le code suivant :

```
class B : public A1 {
protected: A2 v;
};
```

3. constructeur par défaut : Cd de A1 puis Cd de A2 (=constructeur par défaut sur chacun des champs). Rappel : BIT = pas de constructeur.

4. code équivalent :
```
B() : A1(), v() {}
```
5. constructeur par copie : Cc de A1 et Cc de A2.
6. code équivalent :
```
B(const B& b) : A1(b), v(b.v) {}
```
Noter que l'appel A1(b) est un upcasting de B vers A1.
7. assignation par copie : AC= de A1 et AC= de A2.
8. code équivalent :
```
B& operator=(const B& b) {
  if (&b != this) {
        a1 = b.a1; // utilise l'AC de A1
        v = b.v;   // utilise l'AC de A2
    }
}
```
9. destructeur par défaut : Dd de A2 puis Dd de A1 (=destructeur par défaut sur chacun des champs). Remarquer qu'elle a lieu dans l'ordre inverse. Rappel : BIT = pas de destructeur.
10. pas de code équivalent dans ce cas : un champ de A1 a la portée de l'objet B. La portée de la classe héritée est aussi celle de A1 (inclusion directe des champs). Ferait ceci :
```
~B(const B& b) {  // ne jamais écrire ce code
    v.~A2();      // lance le destructeur sur chaque champs
    A1::~A1();    // lance le destructeur de la partie héritée
}
```
11. déplacement : utilise par défaut la copie champs à champs
constructeur par déplacement : constructeur par copie champs à champs,
assignation par déplacement : assignation par copie champs à champs.

Exercice 3: Héritage, interface et virtualité

On reprend les classes définies à l'exercice 1 (`Point2D`, `Stroke`, `Circle`) afin de les enrichir.

1. On commence tout d'abord par les définitions de base :
   (a) Définir la structure simple `Point2D` destinée à représenter un point en 2D avec un constructeur par défaut, par copie et par coordonnées, et un setter permettant de fixer le position du point.
   (b) Définir la classe de base `Stroke` destinée à représenter tout type de traits, et a pour but de devenir une classe abstraite. Elle contiendra un point `start` représentant la position du trait, un constructeur par défaut, par copie, à partir d'un point et un destructeur.
   (c) Si la classe est abstraite, sert-il à quelque chose de définir un constructeur par copie ?
   (d) Définir la classe `Circle` héritant de la classe `Stroke`, définit par un centre (point `start` hérité) et son rayon. On donnera 3 constructeurs : par copie, avec rayon seul (centre par défaut), avec rayon + centre.
   (e) Définir la classe `Segment` héritant de la classe `Stroke`, définit par un début (hérité) et une fin (`end`). On donnera 2 constructeurs : par copie et à partir de deux points.
2. On veut ajouter un méthode virtuelle `length()` à la classe `Stroke` permettant de calculer la longueur du trait sur tout objet `Stroke` concret.

(a) Pourquoi `length()` transforme-t-elle `Stroke` en une classe abstraite ?
(b) Ajouter la définition de `length()` à `Stroke`.
(c) Ajouter la définition de `length()` à `Circle`.
(d) Ajouter la définition de `length()` à `Segment`.

3. On veut maintenant ajouter une interface `iTransform` permettant la transformation géométrique (translation et rotation) de tout type de trait.
   (a) Quelle est la particularité d'une interface ?
   (b) Définir l'interface.
   (c) Comment utiliser cette interface ?
   (d) On voudrait aussi appliquer les mêmes opérations géométriques sur un `Point2D`, pourquoi n'est-ce-pas une bonne idée d'utiliser cette interface dans cette classe ?
   (e) Implémenter la translation et la rotation pour un `Point2D`.
   (f) Ajouter la fonctionnalité à `Stroke`.
   (g) Ajouter la fonctionnalité à `Circle`.
   (h) Ajouter la fonctionnalité à `Segment`.
4. On voudrait maintenant faire que la surcharge de l'opérateur `operator<<` sur le flux de sortie pour l'objet sous toute ses formes. A savoir, on voudrait écrire :

```
Stroke *s = new Circle(4.f);
cout << s;  // ceci affiche les caractéristiques de Circle
```

   (a) Définir la surcharge de l'opérateur `operator<<` pour `Point2D`.
   (b) Pourquoi n'est il pas possible de simplement rendre `operator<<` virtuel ?
   (c) Pourquoi est-il absolument nécessaire de passer par une méthode virtuelle pure ?
   (d) Comment faire en sorte que la surcharge de l'opérateur `operator<<` exécute l'affichage approprié ?
   (e) Effectuer l'implémentation pour la classe `Stroke`.
   (f) Effectuer l'implémentation pour la classe `Circle`.
   (g) Effectuer l'implémentation pour la classe `Segment`.
5. On voudrait maintenant ajouter un compteur privé à la classe `Circle` afin de compter le nombre d'objets créés.
   (a) Comment réaliser ceci en transformant la classe `Circle` ?
   (b) L'implémenter.
   (c) Existe-t-il des cas où le compteur ne donnera pas des résultats corrects ?
   (d) Indiquer comment corriger ce problème.

Solution:

1. définition de base
   (a) code `Point2D` de base :

```
struct Point2D  {
  float x, y;
  Point2D(float u, float v) : x(u), y(v) {}
  Point2D() : Point2D(0.f, 0.f) {};
  Point2D(const Point2D& p) : x(p.x), y(p.x) {} // par défaut
  inline void set(float u, float v) { x = u; y = v; }
};
```

   (b) code `Stroke` de base :

```
class Stroke {
  protected: Point2D start;
  public:
    Stroke(const Point2D &c) : start(c) {}
    Stroke() : start() {}
    Stroke(const Stroke &c) : start(c.start) {} // par défaut
    ~Stroke() {}  // par défaut
};
```

(c) Réponse naïve : non, car l'objet étant abstrait, il ne sera jamais copié.
Réponse éclairée : oui, peut être utilisée pour copier la partie `Stroke` d'un objet concret ayant hérité de `Stroke`.

(d) code `Circle` de base :

```
class Circle : public Stroke {
protected:
  float radius;
public:
  Circle(const Point2D &c, const float r) : Stroke(c), radius(r) {}
  Circle(const float r) : Stroke(), radius(r) {}
  // écriture 1: utilise le constructeur avec le centre
  Circle(const Circle& c) : Stroke(c), radius(c.radius) {} // par défaut
  // ou Circle(const Circle& c) : Stroke(c.start), radius(c.radius) {}
  ~Circle() {} // par défaut
};
```

(e) code `Segment` de base :

```
class Segment : public Stroke {
protected: Point2D end;
public:
  // écrire 2: utilise le constructeur par copie de Stroke
  Segment(const Segment &s) : Stroke(s), end(s.end) {}
  Segment(const Point2D& p1, const Point2D& p2) : Stroke(p1), end(p2) {}
};
```

2. méthode virtuelle `length`

(a) C'est une méthode virtuelle pure, impossible de calculer la longueur sans connaître l'objet concret. L'ajout d'une méthode virtuelle pure à une classe transforme la classe en une classe abstraite.

(b) ajouter à la classe `Stroke`

```
public: virtual float length() = 0;
```

(c) ajouter à la classe `Circle`

```
public: virtual float length() { return 2.f*3.14f*radius; }
```

(d) ajouter à la classe `Segment`

```
public: virtual float length() { return dist(start, end); }
/// fonctions externes pour cette évaluation
inline float sqrf(float x) { return x*x; }
float dist(const Point2D& p1, const Point2D& p2) {
  return sqrtf(sqrf(p1.x - p2.x) + sqrf(p1.y - p2.y));
}
```

3. interface `iTransform`

(a) une interface ne contient que des méthodes virtuelles pures.

(b) soit le code

```
using Vector = Point2D;
class iTransform {
public:
  virtual void Translate(const Vector &t) = 0;
  virtual void Rotate(const float th) = 0;
};
```

L'hérita

de `iTransform` a pour sens : implémente l'interface.

(c) hériter de l'interface et donner les implémentations des méthodes virtuelles dans la classe concrète.

(d) parce que `Point2D` est :

- une petite structure, la rendre virtuelle ajoute une VTABLE ce qui double la taille de la structure.
- une structure de base, probablement utilisée lors de calculs intensifs (ajout d'indirections sur l'appel des transformations si le type réel de l'objet ne peut pas être déterminé à la compilation et conduit à un lien tardif) ou stockés en très grand nombre (ajout d'un pointeur vers la VTABLE).

(e) ajouter à la classe `Point2D`

On n'hérite pas de `iTransform` mais on implémente les méthodes.

```
inline void Translate(const Point2D& p) { x += p.x; y += p.y; }
inline void Rotate(const float th) {
  set(x * cos(th) - y * sin(th), x * sin(th) + y * cos(th));
}
```

(f) ajouter à la classe `Stroke`

```
class Stroke : public iTransform {
public:
  /// implémentation par défaut iTransform2D
  void Translate(const Vector &t) { start.Translate(t); }
};
```

Attention : ces méthodes ne peuvent pas être inline que la virtualité exige l'utilisation d'un pointeur de fonction.

(g) ajouter à la classe `Circle`

```
class Circle : public Stroke {
public:
  // iTransform
  // Translate: cell de Obj2D
  void Rotate(const float th) {}
};
```

(h) ajouter à la classe `Segment`

```
class Segment : public Stroke {
public:
  // iTransform
  void Translate(const Vector &t) {
    start.Translate(t);
    end.Translate(t);
  }
  void Rotate(const float th) {
    start.Rotate(th);
    end.Rotate(th);
  }
};
```

4. surcharge <<

   (a)
```
struct Point2D  {
  float x, y;
  friend ostream& operator<<(ostream &os, Point2D& p) {
    return os << "(" << p.x << "," << p.y << ")";
  }
};
```

   (b) parce que `operator«` est une fonction externe et non une méthode. Il n'est donc pas possible d'utiliser la RTTI pour appeler automatiquement la fonction d'affichage sur le type sous-jacent de l'objet. Noter aussi que la surcharge devra concerner des pointeurs car les objets concrets ne sont pas par essence polymorphes.

   (c) c'est le seul outil qui existe permettant de déterminer automatiquement le type sous-jacent d'un objet.

   (d) définir une méthode virtuelle pure intermédiaire (qui définit comment envoyer un pointeur sur un objet de ce type dans un flux), qui sera appelée lors de l'appel de `operator«`.

   (e) ajouter à la classe `Stroke`

```
class Stroke : public iTransform {
    protected: virtual ostream& view(ostream&) const  = 0;
  // Obj *o car Obj étant un objet abstrait impossible de l'instancier
  friend ostream& operator<<(ostream &os, const Stroke *o) {
    return o->view(os);
  }
};
```

   (f) ajouter à la classe `Circle`

```
class Circle : public Stroke {
protected:
  virtual ostream& view(ostream& os) const {
    return os << "Circle(center=" << start << ",radius=" << radius << ")";
  }
    friend ostream& operator<<(ostream &os, const Circle &c)
      { return c.view(os); }
}
```

   (g) ajouter à la classe `Segment`

```
class Segment : public Stroke {
protected:
  virtual ostream& view(ostream& os) const {
    return os << "Segment(P1=" << start << ",P2=" << end << ")";
  }
    friend ostream& operator<<(ostream &os, const Segment &c)
    { return c.view(os); }
}
```

5. compteur d'objet circle

   (a) Ajouter une variable statique, l'incrémenter dans les constructeurs, la décrémenter dans le destructeur.

   (b) Ajouter à la classe `Circle`

```
class Circle : public Stroke {
private: static int  oCnt;
public:
  Circle(const Point2D &c, const float r) : Stroke(c), radius(r) { oCnt++; }
  Circle(const float r) : radius(r) { oCnt++;  }
  // écriture 1: utilise le constructeur avec le centre
  Circle(const Circle& c) : Stroke(c.start), radius(c.radius) { oCnt++; }
  ~Circle() { oCnt--;  }
  // compteur
  static int Count() { return oCnt; }
};
int Circle::oCnt = 0;
```

(c) Oui, si un objet est détruit alors qu'il est sous une forme polymorphe, son constructeur ne sera pas exécuté.

```
Stroke  *s = new Circle(1.f); // constructeur lancé => oCnt++
delete s; // s détruit comme un Stroke, ~Circle non exécuté.
```

(d) faire en sorte que tous les destructeurs soit virtuels.

```
// dans Stroke:
class Stroke : public iTransform {
   public: virtual ~Stroke() = 0
}
inline Stroke::~Stroke() = default;
// dans Circle:
class Circle : public Stroke {
   public: virtual ~Circle() { oCnt--;  }
}
```