Exercice 4: Classe `KeyId`

On veut définir une classe destinée à stocker un couple (Key,UniqueId) telle que UniqueId est un identificateur unique pour chacun des objets créés (y compris lorsqu'on le copie) et Key est une clef associée à cet identificateur. Dans un premier temps, on souhaite que l'UniqueId ne puisse pas être séparé de l'objet avec lequel il a été créé.

1. Comment faire en sorte que ces contraintes soient respectées lors de l'écriture de la classe ?
2. Déclarer la classe (constructeur à partir d'une clef, set/getter, constructeur et assignation par copie).
3. Définir le constructeur à partir d'une clef.
4. Définir les setter/getters.
5. Définir le constructeur par copie.
6. Définir l'assignation par copie.
7. Doit-on faire quelque chose pour le constructeur et l'assignation par déplacement ?
8. Quel est l'inconvénient d'une telle approche ?
9. Quel serait le sens d'une construction ou d'une assignation par déplacement ?

Solution:

1. définir un static qui va être utilisé pour produire les identificateurs uniques lors des constructions, et faire en sorte que toute opération de copie ne permette pas la copie de l'identifiant.
2. voir le code suivant : le question 2 demande juste la déclaration (prototypes), les suivantes le code.

```
class KeyId {                                                    // Q1
private:                                                         // Q1
    uint32_t            UniqueId;                                // Q1
    uint32_t            Key;                                     // Q1
    static uint32_t     LastId;                                  // Q1 + ci-après
public:                                                          // Q1
    KeyId(uint32_t k) : Key(k), UniqueId(LastId++) {}            // Q1+Q3
    inline uint32_t get_Key() const { return Key; }              // Q1+Q4
    inline uint32_t get_Id() const { return UniqueId; }          // Q1+Q4
    inline void set_Key(uint32_t k) { Key = k; }                 // Q1+Q4
    // Q3: pas de setter pour l'UniqueId car il ne doit pas changer
    KeyId(const KeyId &k) : Key(k.Key), UniqueId(LastId++) {}    // Q1+Q5
    KeyId& operator=(const KeyId &k) {                           // Q1+Q6
        if (this != &k) Key = k.Key;  // on conserve l'Id
        return *this;
    }
    // Q6: oui il faut faire quelque chose, sinon les objets temporaires
    //     peuvent créer et copie des UniqueId
    KeyId(KeyId &&k) = delete;                                   // Q7
    KeyId& operator=(KeyId &&k) = delete;                        // Q7
    friend ostream& operator<<(ostream& os, const KeyId& a);     // pour tests
};
```

```
uint32_t    KeyId::LastId = 0;                                   // Q1 !!!!
// déclaration externe à la classe
ostream& operator<<(ostream& os, const KeyId& a) {
  os << "(" << a.UniqueId << "," << a.Key << ")";
  return os;
}
```

8. L'objet ne peut jamais être passé par copie ou retourné par copie, donc toujours passé par référence ou pointeur (interdit la production d'objet temporaire de type KeyId).

9. Il serait possible d'avoir une autre approche : celle d'autoriser les objets temporaires, et de voler les identifiants des rvalues. Souci : un cast en rvalue (std : :move) permettrait de voler un identifiant.

```
// le constructeur par déplacement vole la valeur de l'identifiant
KeyId(KeyId &&k) : Key(k.Key), UniqueId(k.UniqueId) {}
// l'assignation par déplacement vole la valeur de l'identifiat
KeyId& operator=(KeyId &&k) {
    if (this != &k) { Key=k.Key; UniqueId=k.UniqueId; };
    return *this;
}
```

Exercice 5: Littéraux

On veut définir une classe Angle tel qu'elle est utilisée pour stocker un angle en radian (unité par défaut).

1. écrire la classe Angle.
2. peut-on construire directement un Angle depuis un double (i.e. écrire Angle a = 0.4) ?
3. ajouter une surcharge de l'opérator << sur le flux de sortie standard afin que son unité soit affichée.
4. modifier la classe permette une conversion direct en double (i.e. sans utiliser de getter),
5. ajouter une méthode permettant d'obtenir l'angle en degré.
6. on veut utiliser les littéraux _deg et _rad afin d'initialiser un angle soit à partir de constante double en degré ou en radia. Écrire les fonctions nécessaires.
7. donner un code qui illustre les fonctionnalités décrites ci-dessus.

Solution:

1. classe `Angle` :

```
// must be before anything else
const double     Pi_d = M_PI;
const float      Pi = static_cast<float>(M_PI);
```

```
class Angle {
private:
  double value;   // stocké en radian
public:
  Angle(double v) : value(v) {}
  // conversion d'un Angle en double: pas besoin de getter! (question 4)
  operator double() const { return value; }
  // l'angle en degré (question 5)
  double deg() { return value * 180.0 / Pi_d; }
  // surcharge pour l'affichage (question 3)
  friend ostream& operator<<(ostream& os, const Angle& a);
};
```

2. le constructeur le permet directement.
3. surcharge de l'opérator <<

```
// question 3
ostream& operator<<(ostream& os, const Angle& a) {
  os << a.value << "rad";
  return os;
}
```

4. littéraux

```
// question 6
Angle operator"" _rad(long double x) {
    return Angle(static_cast<double>(x)); }
Angle operator"" _deg(long double x) {
    return Angle(static_cast<double>(x / 180.L*Pi_ld)); }
```

5. code d'illustration

```
// question 7
Angle a = Pi_d / 2.0;
cout << "a=" << a << endl;
Angle b = 180.0_deg;    // note 180_deg ne fonctionne pas car 180 est un entier
cout << "b=" << b << endl;
Angle c = 180.0_rad;
cout << "b=" << b << endl;
double V1 = a;
double V2 = 3.0 * a; // si conversion en float aussi définie, ceci devient ambigu
double V3 = a.deg() / 6;
cout << "V1=" << V1 << " V2=" << V2 << " V3=" << V3 << endl;
```