Introduction aux classes en C++

Loïc Joly, Michel Simatic et Amina Guermouche

Télécom SudParis

14 mai 2025

Plan

1. Les classes
2. Surcharge des opérateurs
3. Mot-clé explicit

Les classes

1. → Les classes
2. Surcharge des opérateurs
3. Mot-clé explicit

Définition d'une classe

```
class Toto{
 private:
  // Liste d'attributs
 public:
  // Liste des fonctions
};
```

à mettre dans un fichier `.h` (ou `.hpp`) pour les déclarations et dans un fichier `.cpp` pour les définitions.

Constructeurs

- Constructeur par défaut : à **ne** déclarer, voire définir **que** lorsque la valeur par défaut a un sens !

```
class Toto{
 private:
  // Liste d'attributs
 public:
  Toto(){}
};
```

- NB : Tout ce qui n'est pas initialisé par le constructeur prend les valeurs par défaut données par C++.
- Les autres constructeurs ont un ou plusieurs paramètres :

```
class Toto{
 private:
  int att1;
 public:
  Toto(int attribut1) : att1(attribut1){}
}
```

Surcharge des opérateurs

1. Les classes
2. → Surcharge des opérateurs
3. Mot-clé explicit

Surcharge des opérateurs (1/2)

- Quand on écrit : `a @ b`, le compilateur comprend en fonction de comment l'opérateur est défini :
  - `a.operator@(b)` : Notion de *Fonction membre* de la classe.
  - `operator@(a,b)` : Notion de *Fonction libre* de la classe.
- Vaut-il mieux implémenter un opérateur comme une *Fonction membre* ou une *Fonction libre* ?
  - Soit une méthode `Polynome(double d);` qui permet d'écrire : `Polynome p{3}`
    - Si nous écrivons `Polynome p2 = p + 3`, alors le compilateur comprend :
      - *Fonction libre* : `operator+(p, 3)`, soit `operator+(p, Polynome(3))` ⇒ OK
      - *Fonction membre* : `p.operator+(3)`, soit `p.operator+(Polynome(3))` ⇒ OK
    - Si nous écrivons `Polynome p2 = 3 + p`, alors le compilateur comprend :
      - *Fonction libre* : `operator+(3, p)`, soit `operator+(Polynome(3), p)` ⇒ OK
      - *Fonction membre* : `3.operator+(p)` ⇒ **KO !**
- Dans un opérateur `+` ou `-`, les 2 opérandes ont le même rôle ⇒ Idéal pour *Fonction libre* !
- Opérateur `=` (copie) n'est pas symétrique ⇒ *Fonction membre* !

Surcharge des opérateurs (2/2)

- Exemple de *Fonction libre*

```
class Carte{
  couleur_t couleur; // Trefle, carreau, coeur, pique
  valeur_t  valeur;  // 2, 3 ... 10, Valet, Dame, Roi, As
  friend bool operator==(const Carte &a, const Carte &b); // `friend` permet a l'operateur == de voir les variables d'instance de Carte
};

bool operator==(const Carte &a, const Carte &b) {
    return (a.couleur == b.couleur) && (a.valeur == b.valeur);
}

int main() {
  Carte a;
  const Carte b;
  bool c = (b == a);
}
```

- Selon SonarQube, un "ami caché" est désormais préféré à une "fonction libre" (cf. [cpp:S2807]).
- Plus d'informations sur la surcharge dans :
  - ce tutoriel,
  - ce cours (qui recommande de surcharger d'abord `+=`, puis `+`, ce qui n'est pas demandé dans le TP CSC4526).

Mot-clé explicit

1. Les classes
2. Surcharge des opérateurs
3. → Mot-clé explicit

Mot-Clé `explicit` : Présentation du problème

- Dans le code suivant, que fait `g(42)` de manière implicite ?

```
class A {
 private:
  std::vector<int> v;
 public:
  A(int taille) : v(taille) {}
};
static void g(A a) {}
int main() {
  g(42);
}
```

- Pour éviter cela, utilisez le mot-clé `explicit` :

```
class A {
 private:
  std::vector<int> v;
 public:
  explicit A(int taille) : v(taille) {}
};
static void g(A a) {}
int main() {
  g(42);   // Erreur de compilation
  g(A{42}); // Compilation OK
}
```

Mot-Clé `explicit` : Conseils d'utilisation

- Conseils (tirés de ce tutoriel)
  - *Make any constructor that accepts a single argument* `explicit` *by default. If an implicit conversion between types is both semantically equivalent and performant, you can consider making the constructor non-explicit (*`explicit(false)`*).*
  - *Do not make copy or move constructors explicit, as these do not perform conversions.*
- Dans le cas du TP Polynome :
  - Écrire `explicit(false) Polynome(double d);` permet de n'avoir qu'un seul `Polynome operator+(Polynome const& p1, Polynome const& p2);`. En contrepartie, le compilateur crée une instance de `Polynome` pour cet entier, appelle `operator+()`, puis détruit cette instance.
  - Garder `explicit Polynome(double d);` pallie cette contrepartie (donc, meilleurs temps d'exécution), si on accepte de définir en plus `Polynome operator+(Polynome const& p1, double const d2);` et `Polynome operator+(double const d1, Polynome const& d2);`.

Mot-Clé `explicit` : Exemple

```cpp
class C {
 private:
  // Aucune variable d'instance pour simplifier le code
 public:
  C() = default;
  explicit C(double d) {}
  friend C operator+(C const& c1, C const& c2) { return C{}; } // Dans un vrai code, il faudrait calculer les variables d'instance de C{}
                                                               // en fonction des variables d'instance de c1 et de c2.
  friend C operator+(C const& c1, double const d) { return C{}; } // Idem
  friend C operator+(double const d, C const& c2) { return C{}; } // Idem
};

int main() {
  C c;
  auto res1 = c + c;
  auto res2 = c + 1.0;
  auto res3 = 1.0 + c;
}

// Expériences :
// 1) Vérifiez que le code compile.
// 2) Mettez en commentaire les 2 `operator+()` avec `double` et regardez l'erreur de compilation au niveau de `res2` et `res3`.
// 3) Mettez ensuite `explicit(false) C(double d) {}` : Le code compile à nouveau.
// 4) Enlevez `(false)` et remplacez `1.0` par `C{1.0}` : Le code compile aussi.
// 5) Revenez au code initial et remplacez le 3eme `operator+()` par une **declaration** : Le code compile, mais echoue au link.
```

Questions ?

1. Les classes
2. Surcharge des opérateurs
3. Mot-clé explicit