Notes de cours C++

Thibaut Marmey

November 30, 2018

Contents

1 Programmation C++

1.1 L'héritage

- Cours openclassroom : héritage
- Déclaration d'une classe héréditaire :

```
class Classe_fille : public|protected|private Classe_mere1
[, public|protected|private Classe_mere2 [...]]
{
    /* Definition de la classe fille. */
};
```

- les données publiques d'une classe mère deviennent soit publiques, soit protégées, soit privées selon que la classe fille hérite en public, protégé ou en privé.
- La classe fille possède les attributs et les méthodes de la classe mère. Elle possède en plus de cela ses propres attributs et méthodes
- Possibilité de surcharger les méthodes de la classe mère dans la classe fille
- Surcharge du constructeur : on peut appeler le constructeur de la classe mère dans le constructeur de la classe fille.
  Pour déclarer un constructeur dans le .h : il doit avoir le même nom que la calsse, et il ne doit rien renvoyer. *maClasse()*
  On écrira aussi le constructeur dans le .cpp de la manière suivante :
  maClasse::maClasse() : blaba, bla, bla { }

- Masquage de fonctions de la classe mère. On peut substituer le nom d'une fonction présente dans la classe mère et utiliser sous le même nom une méthode spécifique dans une classe fille. On peut toujours appeler la méthode de la classe mère dans la méthode de la classe fille en spécifiant l'appel à la classe mère grâce au double deux points "nomClasseMere::nomMéthode()".

- Dérivation de type : on peut substituer un objet de la classe fille à un pointeur ou une référence vers un objet de la classe mère. On peut affecter un élément enfant à un élément parent
  Il est possible d'écrire *monPersonnage = monGuerrier* car un guerrier est un personnage. L'inverse n'est pas possible ~~*monGuerrier = monPersonnage*~~

- La dériation de type est très pratique dans l'appel d'un élément dans une fonction par exemple. Si l'argument à mettre est de la classe *Personnage* il est alors possible de mettre tous les autres classes filles de *Personnage*
  void coupDePoing(Personnage & cible) const; : *cible* peut être de type *Personnage* ou *Guerrier*

- Le type *protected* permet aux attributs d'être accessible par les classes filles et inaccessible de l'extérieur

1.2 Polymorphisme

- le polymorphisme est un mécanisme dynamique permettant, par voie d'héritage, de spécialiser dans des classes dérivées les comportements annoncés ou implémentés dans des classes de base, indirectes ou non.

- **Résolution statique des liens.** La fonction reçoit un type de data, c'est donc toujours les méthodes de ce type qui sera utilisée. C'est donc le type de la variable qui détermine quelle fonction membre appeler et non sa vraie nature.
  Ex pratique :
  void presenter(Vehicule v) //Présente le véhicule passé en argument
  { v.affiche(); }
  La fonction reçoit un véhicule (classe mère) c'est donc les méthodes de véhicule qui sont appelées même si une surcharge de méthode est présente dans la classe fille.

- **Réolution dynamique des liens.** Lors de l'exécution le programme utilise la bonne version des méthodes car il sait si l'objet est de type mère ou type fille.
  - utiliser un pointeur ou une référence
  - utiliser des méthodes virtuelles

- Il faut placer un pointeur ou une référence comme argument dans la fonction *void presenter(Vehicule const& v)*

- Rajouter *virtual* devant la méthode dans la classe mère (seulement dans le .h) et c'est optionnel dans les classes filles.

1.3 Design Pattern

- Ce sont des modèles théoriques adaptables qui résolvent un problème précis.
- **Un prototype :** un prototype est une classe dont le but est d'être clonée.
- **Le singleton :** permet de s'assurer qu'il n'existe qu'une unique instance d'une classe donnée.
  Est une variable globale.
- **La fabrique :** classe dont le rôle est de créer d'autres objets.
- **Les décorateurs :** sont l'ensemble des classes permettant d'étendre dynamiquement le rôle d'une classe de base.

1.4 Template

- Les templates sont des fonctions spéciales qui peuvent être utilisées avec des types génériques. Cela nous permet de créer une fonction template dont l'utilisation n'est pas réstreinte à un seul type de données, sans répéter le code entier pour chaque type.

```
template <class myType>
myType GetMax (myType a, myType b) {
 return (a>b?a:b); }
```

- Pour utiliser les fonctions templates on utilise le schéma suivant :

```
//function_name <type> (parameters);
int x,y;
GetMax <int> (x,y);
```

- Utilisation des templates avec les classes :

```
template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)
      {a=first; b=second;}
    T getmax ();
};

template <class T> //T for template parameter
T mypair<T>::getmax () //1st T for the type return by the function
//2nd T for requirement to specify the function's template parameter
{
  T retval;
  retval = a>b? a : b;
  return retval;
}
```

1.5 Autre

- Utilisation de = dans un *if*.

```
int a = 1
int b = 2;
if ((a=b)) {...} //is true
blabla...
int b = 0;
if ((a=b)) {...} //is false
```

- Utilisation de *auto* (C++11) : cet outil permet de spécifier automatiquement le type de la variable en jeu.
  ex : auto x = 1; auto y = 3.1; auto z = 'a';
  output : x est int, y est double, z est char

- *Lambda* function : *ex : auto allowed = [&](int x, const std::vector⟨int⟩ &vect)*{...}
  - a. [=] capture all variables within scope by value
  - b. [&] capture all variables within scope by reference
  - c. [& var] capture var by reference
  - d. [&, var] specify that the default way of capturing is by reference and we want to capture var
  - e. [=, & var] capture the variables in scope by value by default, but capture var using reference instead