Introduction à l'héritage en C++

L3 math informatique S6

26 février 2025

Principes de l'héritage

- Permet de modéliser une relation **is-a**.
- Facilite la réutilisation du code.
- Permet le polymorphisme (via fonctions virtuelles).

Exemple fil rouge

Nous allons utiliser l'exemple suivant : une classe `Animal` et ses sous-classes `Chien` et `Chat`.

Héritage simple

Déclaration d'une classe de base :

```
class Animal {
public:
    void parler() { std::cout << "Je␣suis␣un␣animal." << std::endl; }
};
```

Classe dérivée et constructeurs

```
class Chien : public Animal {
public:
    Chien() { std::cout << "Chien␣créé." << std::endl; }
    ~Chien() { std::cout << "Chien␣détruit." << std::endl; }
};
```

Les constructeurs sont appelés dans l'ordre de dérivation.

Appel des constructeurs

```
int main() {
    Chien c; // Affiche "Chien créé."
} // Affiche "Chien détruit."
```

Héritage multiple (à éviter)

Problèmes potentiels :

- Ambiguïtés (deux classes mères ayant une même méthode).
- Gestion complexe des constructeurs.

Exemple d'héritage multiple

```
class Animal { };
class Robot { };
class ChienRobot : public Animal, public Robot { };
```

Risques : Ambiguïté si `Animal` et `Robot` ont des méthodes identiques.

Fonctions virtuelles et polymorphisme

```
class Animal {
public:
    virtual void parler() { std::cout << "Je␣suis␣un␣animal." << std::endl;
         }
};
class Chien : public Animal {
public:
    void parler() override { std::cout << "Ouaf␣!" << std::endl; }
};
```

La méthode parler() est virtuelle dans `Animal` et redéfinie dans `Chien`. Le mot-clé `override` est utilisé pour indiquer que la méthode redéfinit une méthode virtuelle (indispensable depuis C++11)

Appel d'une fonction virtuelle

```
Animal* a = new Chien();
a->parler(); // Affiche "Ouaf !" grâce au polymorphisme
delete a; // Nettoyage
```

Ajout d'une classe Chat

```
class Chat : public Animal {
public:
    void parler() override { std::cout << "Miaou ! " << std::endl; }
};
```

Comme pour Chien, la méthode parler() est redéfinie.

Utilisation de la classe de base

```
int main() {
    Animal a;
    a.parler(); // Affiche "Je suis un animal."
}
```

La classe de base peut être utilisée seule, sans héritage.

Définition d'une fonction virtuelle pure

```
class Animal {
public:
    virtual void parler() = 0; // Fonction virtuelle pure
};
```

Une classe contenant au moins une fonction virtuelle pure est une classe abstraite.

Implémentation dans les classes dérivées

```
class Chien : public Animal {
public:
    void parler() override { std::cout << "Ouaf␣!" << std::endl; }
};

class Chat : public Animal {
public:
    void parler() override { std::cout << "Miaou␣!" << std::endl; }
};
```

Les classes dérivées doivent implémenter la fonction virtuelle pure, sinon elles seront aussi abstraites.

Utilisation du polymorphisme

```
int main() {
    Animal* a1 = new Chien();
    Animal* a2 = new Chat();

    a1->parler(); // Affiche "Ouaf !"
    a2->parler(); // Affiche "Miaou !"

    delete a1;
    delete a2;
}
```

L'utilisation d'un pointeur sur la classe de base permet un comportement polymorphique.

Problème de troncature (slicing) I

```
class Animal {
public:
    virtual void parler() { std::cout << "Je␣suis␣un␣animal." << std::endl;
         }
};

class Chien : public Animal {
public:
    void parler() override { std::cout << "Ouaf␣!" << std::endl; }
};

int main() {
    Animal a1 = Chien(); // Troncature : a1 devient un simple Animal
    a1.parler(); // Affiche "Je suis un animal." et non "Ouaf !"
}
```

L'affectation copie uniquement la partie `Animal`, perdant ainsi les spécificités de `Chien`.