

# L'héritage

## Introduction au C++ et à la programmation objet

E. Courcelle

CALMIP, URA 3669

Mai 2022

## 1 Classes abstraites et concrètes

## 2 Le polymorphisme

## Une classe abstraite

Une machine à café, ça... fait le café

...mais on ne précise pas comment. *Version diagramme*

## Cafetiere

- eau : ReservoirEau
- gobelets : ReservoirGobelets
- cuillers : ReservoirCuillers
- sucre : ReservoirSucre

```
+ faire_le_cafe() : void
+ lire_etat() : void
# verser_eau() : void
# sucrer() : void
# donner_gobelet() : void
# donner_cuiller() : void
```

# Une classe abstraite

Une machine à café, ça... fait le café

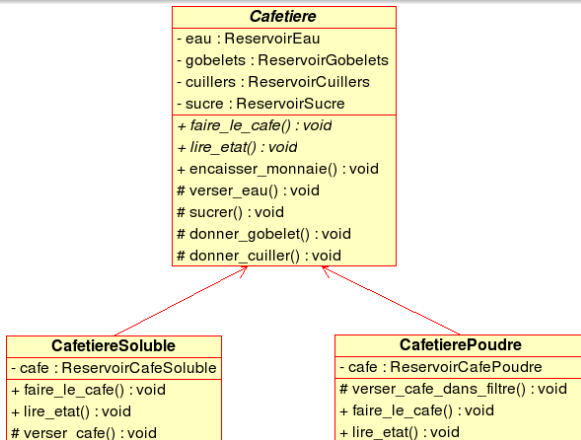
...mais on ne précise pas comment. *Version code*

```
class Cafetiere {  
    public:  
        virtual void faire_le_cafe() = 0;  
        virtual int lire_etat() = 0;  
        void encaisser_monnaie();  
  
    protected:  
        void verser_eau();  
        void sucre();  
        void donner_gobelet();  
        void donner_cuiller();  
  
    private:  
        ReservoirEau eau;  
        ReservoirSucre sucre;  
        ReservoirGobelets gobelets;  
        ReservoirCuillers cuillers;  
}
```

# Plusieurs classes concrètes

## Plusieurs manière de faire le café

...Les classes concrètes précisent les choses. *Version diagramme*



# Plusieurs classes concrètes

## Plusieurs manière de faire le café

...Les classes concrètes précisent les choses. *Version code*

```
class CafetiereSoluble: public Cafetiere {
public:
    void faire_le_cafe() override;
    int lire_etat() override;

protected:
    void verser_cafe();

private:
    ReservoirCafeSoluble cafe;
}

class CafetierePoudre: public Cafetiere {
public:
    void faire_le_cafe() override;
    int lire_etat() override;

protected:
    void verser_cafe_dans_filtre();

private:
    ReservoirCafePoudre cafe;
}
```

# Sections private, protected, public

## private

- Données encapsulées par l'objet
- Fonctions membres correspondant au fonctionnement interne de l'objet.

*On peut les modifier comme on veut !*

## protected

- Pas de données
- Fonctions membres accessibles par les objets dérivés, **et seulement eux**.

*Attention aux modifications, risque de casser le code des classes dérivées (mais ça reste relativement localisé)*

## public

- Pas de données
- Fonctions membres accessibles par le code utilisateur de l'objet.

*ON NE CHANGE RIEN, risque de casser plein de code utilisateur*

# Constructeurs par défaut

```
CafetiereSoluble ma_cafetiere;
```

## Que se passe-t-il ?

- 1 Allocation de mémoire pour Cafetiere **et** CafetiereSoluble
- 2 Appel du **constructeur par défaut de** Cafetiere
- 3 Appel du **constructeur par défaut de** CafetiereSoluble



# Passage de paramètres aux constructeurs

## Un constructeur peut en appeler un autre

Le constructeur de la classe dérivée **appelle explicitement le constructeur de la classe de base**.

- 1 Allocation de mémoire pour Cafetiere et CafetiereSoluble
- 2 Appel du **constructeur de Cafetiere avec ses paramètres**
- 3 Initialisation des membres de CafetiereSoluble
- 4 Appel du code du constructeur de CafetiereSoluble

```
class Cafetiere {  
public:  
    Cafetiere(float e, int g, int c, float s) : eau(e), gobelets(g),  
        cuillers(c), sucre(s) {};  
    ...  
}
```

```
class CafetiereSoluble: public Cafetiere {  
public:  
    CafetiereSoluble(float e, int g, int c, float s, float f) :  
        Cafetiere(e,g,c,s), cafe(f) {};  
    ...  
}
```

# Le destructeur

C'est pareil, sauf que c'est l'inverse

- 1 Appel du code du destructeur de CafetiereSoluble
- 2 Appel du code du destructeur de Cafetiere
- 3 Désallocation de la mémoire

...mais c'est comme les antibiotiques: c'est pas automatique !

Il faut définir pour chaque classe de la hiérarchie un destructeur virtuel. Il peut ne rien faire ({}), mais il garantit que le destructeur de la classe de base est bien appelé.

```
class Cafetiere {  
public:  
    virtual ~Cafetiere() {};  
    ...  
}  
classe CafetiereSoluble: public Cafetiere {  
    ~CafetiereSoluble() {};  
}
```

# Déclarations

On peut pas !

```
Cafetiere C;
```

On peut !

```
Cafetiere * C;
```

On peut !

```
void ma_fonction(Cafetiere& c);
```

```
CafetiereSoluble cs;
```

```
CafetierePoudre cp;
```

```
ma_fonction(cs);
```

```
ma_fonction(cp);
```

# Un vecteur polymorphe

## Une cafeteria

Le code ci-dessous permet de déclarer un vecteur (tableau) polymorphe, c'est-à-dire un vecteur contenant des objets de plusieurs types. La notation est un peu lourde, elle fait appel à la stl, **mais c'est la bonne manière de faire.**

```
vector<shared_ptr<Cafetiere>> machines;  
machines.push_back(make_shared<CafetiereSoluble>());  
machines.push_back(make_shared<CafetiereSoluble>());  
machines.push_back(make_shared<CafetierePoudre>());  
  
for (auto m : machines) {  
    if (m->lire_etat() == true) {  
        m->faire_le_cafe();  
        m->encaisser_monnaie();  
    };  
};
```

# Le clonage polymorphique

## Sémantique d'entité

Les objets dont nous parlons ont une sémantique d'entité, ils n'ont pas d'operator=, par contre ils peuvent implémenter le clonage polymorphique.

```
class Cafetiere {  
    ...  
    virtual Cafetiere * Cafetiere clone() = 0;  
}  
  
class CafetiereSoluble {  
    ...  
    CafetiereSoluble * Cafetiere clone() override  
    {  
        return new CafetiereSoluble(*this);  
    }  
};
```