```
class Vecteur {
    ...
public:
    void afficher ();
};
```

```
class Vecteur_color : public Vecteur
{
private:
    char _color[3]; // RGB
public:
    void afficher ();
};
```

$\subseteq$

une sorte de

Polymorphisme → objets ayant type stat / type dyn ($\neq$)

- type stat → déf / déclaré
- type dyn → contenu / ce que contient la var.

type dynamique ≡ pointeur ou réf.

pointeur ou réf sur classe mère Vecteur ← @ d'une classe fille

est une sorte de

Polymorphisme (ex): ⚠ dynamique → pointeurs / réfs.

```
Vecteur * p = new Vecteur_color (255, 0, 10);
```

p → type stat : Vecteur *

p → type dyn : Vecteur_color * $\neq$

p -> affiche ()
!

→ par défaut : celle du type stat (ici : Vecteur)

→ si la meth. est virtuelle (dans la classe mère) ⇝ celle du type dynamique (Vecteur_color).

ex :

```
main : Vecteur * tab[3]; // tab tableau de 3 ptr Vecteur.
       tab[0] = new Vecteur(10);
       tab[1] = new Vecteur_color(20, 255, 0, 0);   // n, RGB
       tab[2] = new Vecteur(20);
       for (int i=0; i<3; i++)
           tab[i] -> affiche();
```

≠

constr. par défaut

```
Vecteur v = Vecteur_color(20, 250, 10, 2);
```

une sorte de Vecteur

← partie Vecteur affectée — ~~color~~

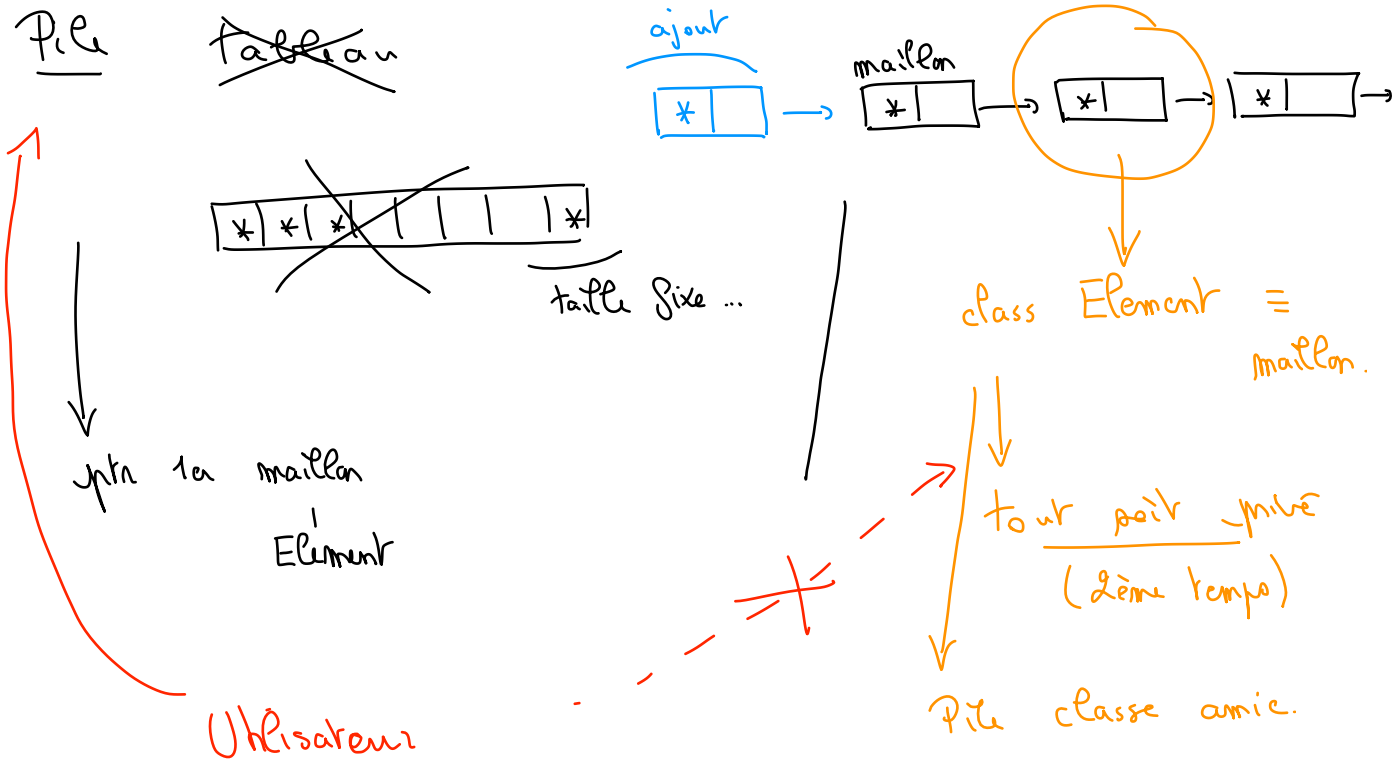
mieux

constr. par copie

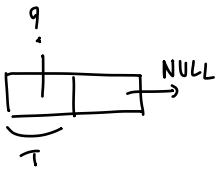
```
Vecteur v(Vecteur_color(20, 250, 10, 2));
```

une sorte de Vecteur

v n'a qu'un type statique → pas de polymorphisme.




```
template <typename T>
class Element {                    T _dat;
                                   Element<T> * _next;
                                   ↘ classe
    // Pas de constr. par défaut
    Element () {}
                ↓
                on ne fait rien mais les données membre
                _dat , _next
                sont construites ss constr. par défaut
                ↓                  ↓
                _dat               _next
                celui de T         celui des ptr & NULL
};
```




```
....
... Pile {
        Element * _data;
        string    _nom;
        int       _taille;
```

```
...
... Elem {
    privée:
        T _dat;
        Element * _suiv;
```

→ pile vide

```
Pile (string s = "toto")
{
    _data = NULL;
    _nom = s;
    _taille = 0;
}
```

← pas de new

push ~ new.

destructeur :

~ Pile ()

↓

détruit tous les maillons

delete _data

↓

⊕ simple en récursif ....

détruire tous les maillons ....

```
Element (T data,
         Element *n = NULL)
{
    _dat = data;
    _suiv = n;
}
```

pas de new

~ Element () {}

pas de delete

_data
_s — string
_taille — int
...

Element * _data ptr 1er maillon.

Element

delete _data

↓

détruit l'élément à l'@ _data

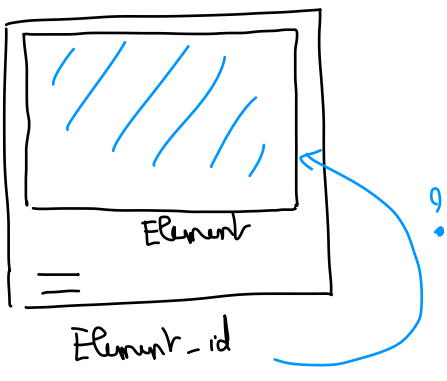
tête ...

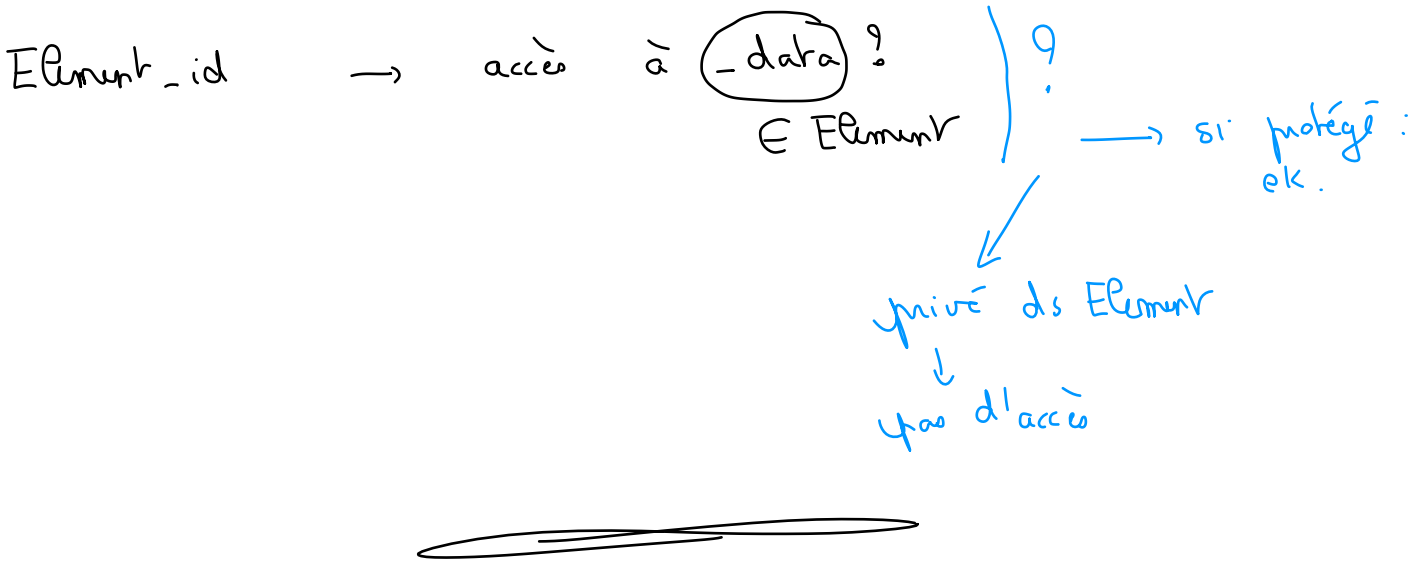
⚠ Element n'est pas 1 classe

↓

Element <T>

```
template <typename T>
Element_id : public Element <T> {
    private :
        int _id;
        static int _cpt;
    :
}
```






```
template <typename Type>
class Element {
private:
        Type _elt ;
        Element<Type> *_next ;

        Element(const Type &) ;
        Element(const Element &) ;
        ~Element() ;
        template <typename T> friend class PileGen ;
};
```