Type statique / dynamique.

→ refs / pointeurs

Vecteur15 : public Vecteur

(sorte de)

ce qui est ds p à 1 instant .... ⇝ varie

```
Vecteur * p = new Vecteur (20);
        ou
      = new Vecteur15;
```

déclaration = Vecteur *

type dynamique : Vecteur *

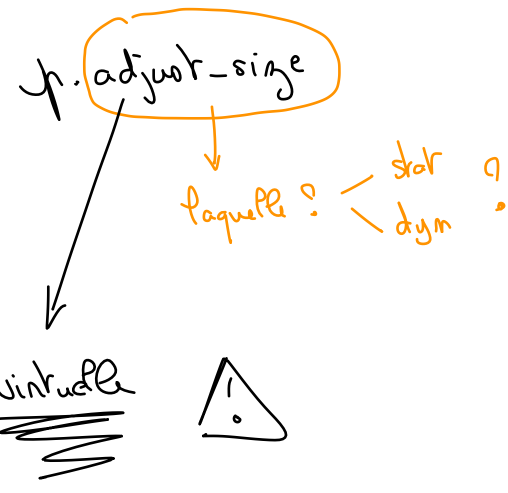
≠

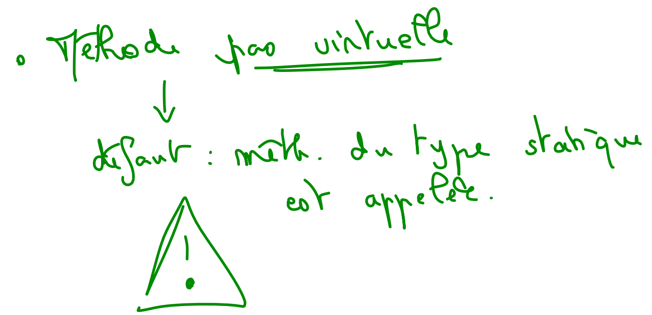
type dynamique : Vecteur15 *

```
Vecteur *p = new Vecteur15
```

p → type stat : Vecteur *

p → type dyn : Vecteur15 *





• méthode virtuelle

↳ type dynamique

```
class Vecteur {
    ⋮
  public:
(*)  • void affiche ();
}
```

• → virtual ou pas

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

surchargée ↓ affiche la couleur en plus.

virtual ⇝ objet dont le type dynamique et statique ≠

ex main

type dynamique → ce qu'il y a dans l'objet à l'instant t

statique → type déclaré

≠ → pointeurs, ref

<u>main</u>:

```
Vecteur * p = new Vecteur_colore (255, 0, 128);
```

Vecteur → type stat

new Vecteur_colore → type dyn.

≠

```
p -> affiche ()
```

quelle méthode ?

(*) → pas de virtual

↳ <u>défaut</u> : méthode du type statique appelée

ici : Vecteur

↓

lol .....

→ si virtual:

↳ type dynamique

<u>ici</u>: vecteur_colore.

<u>ex</u>:

```
Vecteur * tab[3];   // tableau de 3 cases
                                     ↓
                               pointeurs s/ vect.

tab[0] = new Vecteur (20);
tab[1] = new Vecteur_colore (10, 20, 255);
tab[2] = new Vecteur (10);
for (int i = 0 ; i < 3 ; i++)
{
    tab[i] -> affiche ();
}
```

⚠ affiche virtual ds Vecteur

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

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

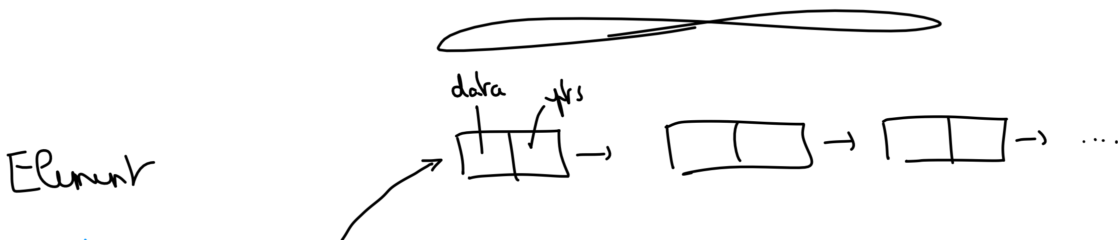
Classes génériques : (template)

.h → déclarations + à la fin #include "....hpp"

.hpp → implémentation
↓
⚠ pas de #include au début !

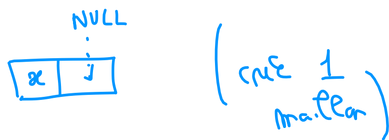
suite du .h ....

Element



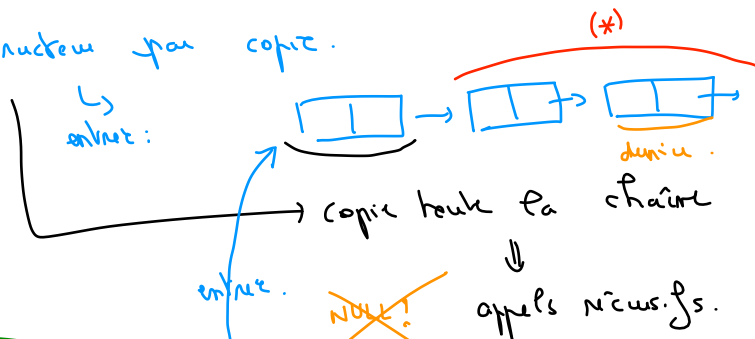
Constructeurs :

1) prend 1 donnée x ⇝



(crée 1 maillon)

2) constructeur par copie.
↳ entrée :



→ copie toute la chaîne
⇓
appels récursifs.

Code à corriger:
e ne doit pas être NULL

Element (const Element & e) // copie le maillon de tête ⊕ copie la suite.

{ ~~if (e ≠ NULL)~~

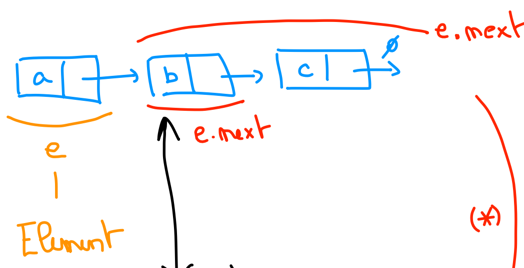
copie du (*)

if (elt.next ≠NULL

```
Element<Type> *tmp ;
tmp = new Element<Type>(elt.next) ; // appel récursif du
constructeur par copie
_next = tmp ;
_elt = elt._elt ;
```

}

in :



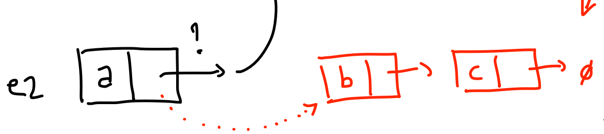
Element e2(e);

↓

e2 copie de e
intégrale ...

✗ pas propre ...

(*) constructeur par copie sur la fin de la liste




```
Element (const Element & e)   → constr. par copie.
{
    Element tmp = Element (e._next);
    this -> _next = &tmp;
```

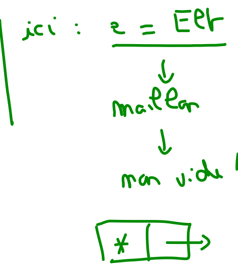
réf.

⚠ var. locale

début à la fin ! (*)

cas terminal ? ↳ liste vide ?

ici : e = Elt → maillon → non vide !



appel réc. .... → cas terminaux ?

```
Element *tmp = new Element (*(e._next));
this -> _next = tmp;
ok  (( this -> _data = e._elt;
```

pointeur s/ élt.

si e._next == NULL

~~*(e._next)~~

cas terminal !

⇓

Code final

```
Element (const Element & e)
{
    this -> _data = e._data;
    if (e._suiv == NULL)
        this -> _suiv = NULL;
    else
    // copier toute la suite ~> constr. par copie.
    {
        this -> _suiv = new Element (*(e._suiv));
    }
}
```