

LANGUAGE OBJ. AV. (C++) MASTER 1

U.F.R. d'Informatique
Université de Paris Cité

Complément sur les classes,

Petit mot sur static :

- le mot clé static (devant la déclaration d'un attribut ou une méthode) exprime qu'il est rattaché directement à la classe (et non pas à chaque objet comme le sont les membres)
- une variable statique est donc de fait partagée
- une fonction statique peut donc être invoquée sans passer par l'intermédiaire d'un objet de la classe

Complément sur les classes,

Petit mot sur static :

```
class X {  
    private :  
        static int x;  
    public :  
        static void f();  
};
```

```
int X::x{0};  
void X::f(){};
```

Le mot clé est utilisé seulement à la déclaration.
Il ne fait pas partie de la signature dans le cpp

Petit mot sur les classes dépendantes

- on ne peut pas écrire :

```
class A {  
    public :  
        B b;  
};
```

```
class B {  
    public :  
        A a;  
};
```

Petit mot sur les classes dépendantes

- on ne peut pas écrire :

```
class A {  
    public :  
        B b;  
};
```

```
class B {  
    public :  
        A a;  
};
```

en effet : on parle de la classe B avant de l'avoir déclarée.

Il faut passer par une "déclaration préalable"

Petit mot sur les classes dépendantes

- on ne peut pas écrire :

```
class B;  
class A {  
    public :  
        B b;  
};  
  
class B {  
    public :  
        A a;  
};
```

en effet : on parle de la classe B avant de l'avoir déclarée.

Il faut passer par une "déclaration préalable"

mais cela reste insuffisant ici ...
intéressons nous aux constructeurs par défaut ...

Petit mot sur les classes dépendantes

- on ne peut pas écrire :

```
class B;  
class A {  
    public :  
        B b;  
        A() : b {} {}  
};  
class B {  
    public :  
        A a;  
        B() : a {} {}  
};
```

en effet : on parle de la classe B avant de l'avoir déclarée.

Il faut passer par une "déclaration préalable"

mais cela reste insuffisant ici ...
intéressons nous aux constructeurs par défaut ...

Ils forment une boucle infinie !

Petit mot sur les classes dépendantes

- définition préalable + pointeur

```
class B;  
class A {  
public :  
    B *b;  
};  
  
class B {  
public :  
    A a;  
};
```

l'utilisation d'un pointeur brise la boucle : le constructeur n'est pas invoqué.

Petit mot sur les classes dépendantes

- définition préalable + pointeur

```
class B;  
class A {  
public :  
    B b;  
};  
  
class B {  
public :  
    A *a;  
};
```

l'utilisation d'un pointeur brise la boucle : le constructeur n'est pas invoqué.

mais vu l'ordre des déclarations on ne peut pas faire dans l'autre sens : les constructeurs existants pour B doivent être connus au moment où la classe A est déclarée

Petit mot sur les classes dépendantes

- définition préalable + pointeur

```
class A;  
class B {  
public :  
    A *a;  
};
```

```
class A {  
public :  
    B b;  
};
```

l'utilisation d'un pointeur brise la boucle : le constructeur n'est pas invoqué.

Petit mot sur les classes dépendantes

- dans le cas d'une compilation séparée

```
// fichier A.hpp
#ifndef _A
#define _A
class B;
class A {
public :
    B *b;
};
#endif
```

```
// fichier B.hpp
#ifndef _B
#define _B
#include "A.hpp"
class B {
public :
    A a;
};
#endif
```

Rappels, vu la semaine dernière :

- Type référence : avec &
- Constructeur par copie : `A (const A &)`
- Destructeur : `virtual ~A()`
- qqs petites choses :
 - `x->m` équivalent à `(*x).m`
 - this est un pointeur
 - const pour méthodes (exemption avec mutable)
- redéfinition de `<<` :
 - `ostream & operator<<(ostream &, const A &)`

Nous allons faire 3 petites études de cas
et discuter un peu copies/destructions

Etude de cas 1 :

destructions problématiques

```
class Segment {
public:
    Segment(Point *, Point *);
private:
    Point *p1, *p2;
};
```

```
Segment::Segment(Point *x, Point *y):p1{x},p2{y} {}
```

```
class Point {
public:
    Point(int x=0,int y=0);
    virtual ~Point();
    int abs, ord;
};
ostream & operator<<(ostream &,
                    const Point &);
```

```
Point::Point(int x, int y):abs{x}, ord{y} {
    cout << "creation de " << *this << endl;
}
Point::~~Point() {
    cout << "destruction de " << *this << endl;
}
ostream & operator<<(ostream & out,
                    const Point & p) {
    out << "(" << p.abs << "," << p.ord << ") ";
    return out;
}
```

```
int main() {
    Segment *s;
    { // un bloc
        Point p0, p1{1,1};
        s=new Segment(&p0,&p1);
    }
    // attention ici s contient 2 liens morts
    cout << *(s->p1) << endl;
}
```

```
creation de (0,0)
creation de (1,1)
destruction de (1,1)
destruction de (0,0)
(0,0)
```

erreur du programmeur, l'exécution se poursuit ... pour l'instant ...

```
class Segment {
public:
    Segment(Point *, Point *);
private:
    Point *p1, *p2;
};
```

```
Segment::Segment(Point *x, Point *y):p1{x},p2{y} {}
```

```
class Point {
public:
    Point(int x=0,int y=0);
    virtual ~Point();
    int abs, ord;
};
ostream & operator<<(ostream &,
                    const Point &);
```

```
Point::Point(int x, int y):abs{x}, ord{y} {
    cout << *this << endl;
}
~Point() {
    cout << *this << endl;
}
ostream & operator<<(ostream & out,
                    const Point & p) {
    out << "(" << p.abs << "," << p.ord << ") ";
    return out;
}
```

il faut quand même le faire exprès ..

```
int main() {
    Segment *s;
    { // un bloc (ou résultat d'une fonction)
        Point p0, p1{1,1};
        s=new Segment(&p0,&p1);
    }
    // attention ici s contient 2 liens morts
    cout << *(s->p1) << endl;
}
```

```
creation de (0,0)
creation de (1,1)
destruction de (1,1)
destruction de (0,0)
(0,0)
```

erreur du programmeur, l'exécution se poursuit ... pour l'instant ...


```
class Segment {  
public:  
    Segment(Point , Point );  
private:  
    Point *p1,*p2;  
};
```

```
Segment::Segment(Point x, Point y):p1{&x},p2{&y} {}
```

petits changements :

- on limite l'usage des pointeurs
- on fait attention à ne pas écrire de bloc ...

```
int main() {  
    Point p0, p1{1,1};  
    Segment s {p0,p1};  
    // ok ??  
    cout << *(s.p1) << endl;  
}
```

```
class Segment {  
public:  
    Segment(Point , Point );  
private:  
    Point *p1,*p2;  
};
```

```
Segment::Segment(Point x, Point y):p1{&x},p2{&y} {}
```



les arguments sont de nouveaux objets
(obtenus par copie)

détruits ici

```
int main() {  
    Point p0, p1{1,1};  
    Segment s {p0,p1};  
    // attention ici s contient 2 liens morts  
    cout << *(s.p1) << endl;  
}
```

```
creation de (0,0)  
creation de (1,1)  
copie de (0,0)  
copie de (1,1)  
destruction de (1,1)  
destruction de (0,0)  
(0,0)  
destruction de (1,1)  
destruction de (0,0)
```

même erreur, on affiche un fantôme ...

```

class Segment {
public:
    Segment(const Point & , const Point & );
    virtual ~Segment();
private:
    Point *p1,*p2;
};

```

```

Segment::Segment(const Point & x, const Point &y)
    :p1{new Point{x.abs,x.ord}},
    p2{new Point{y.abs,y.ord}} {}
Segment::~~Segment() {
    delete p1;
    delete p2;
}

```

```

int main() {
    Point p0, p1{1,1};
    Segment s {p0,p1};
    cout << *(s.p1) << endl;
}

```

```

creation de (0,0)
creation de (1,1)
creation de (0,0)
creation de (1,1)
(0,0)
destruction de (0,0)
destruction de (1,1)
destruction de (1,1)
destruction de (0,0)

```

... on décide de contrôler les copies pour être tranquille.
et on assume la destruction du pointeur dont on est propriétaire
ici plus de pbs ... apparemment ...

```
class Segment {
public:
    Segment(const Point & , const Point & );
    virtual ~Segment();
private:
    Point *p1, *p2;
};
```

```
Segment::Segment(const Point & x, const Point &y)
    :p1{new Point{x.abs,x.ord}},
    p2{new Point{y.abs,y.ord}} {}
Segment::~~Segment() {
    delete p1;
    delete p2;
}
```

```
void f(Segment z) {}

int main() {
    Point p0, p1{1,1};
    Segment s {p0,p1};
    cout << *(s.p1) << endl;
    f(s);
}
```

... on décide de contrôler les copies pour être tranquille.
et on assume la destruction du pointeur dont on est propriétaire
ici plus de pbs ... ??

```
class Segment {
public:
    Segment(const Point & , const Point & );
    virtual ~Segment();
private:
    Point *p1,*p2;
};
```

```
Segment::Segment(const Point
    :p1{new Point{x.abs,x.or
    p2{new Point{y.abs,y.or
Segment::~~Segment() {
    delete p1;
    delete p2;
}
```

le constructeur par copie reprend les même pointeurs que s pour z
 Puis il les détruit qd z est détruit ...
 Puis il affiche un fantôme
 Puis il cherche à les détruire à nouveau, c'est trop ...

```
void f(Segment z) {}

int main() {
    Point p0, p1{1,1};
    Segment s {p0,p1};
    cout << *(s.p1) << endl;
    f(s);
}
```

```
creation de (0,0)
creation de (1,1)
creation de (0,0)
creation de (1,1)
(0,0)
destruction de (0,0)
destruction de (1,1)

RUN FINISHED; Segmentation fault;
```

... on décide de contrôler les copies pour être tranquille.
 et on assume la destruction du pointeur dont on est propriétaire
 ici plus de pbs ... oups !

Moralité :

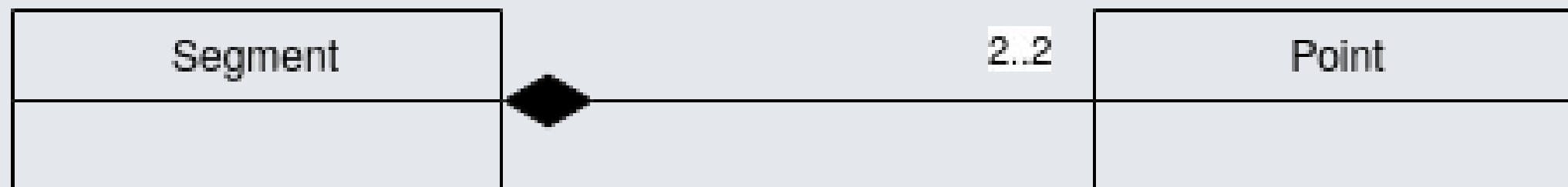
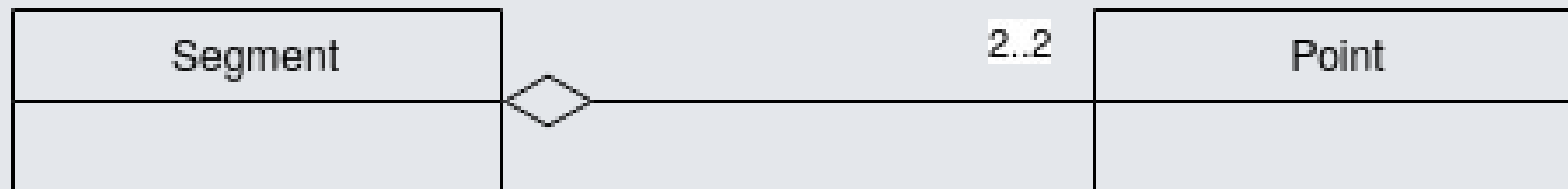
Dès qu'il y a des pointeurs, il faut :

- être sûr que l'objet est bien vivant
- savoir qui a la charge de le détruire

(il y aura fuite de mémoire s'il n'est jamais détruit,
core dumped immédiat si double destruction,
core dumped plus tard si l'espace "fantôme" est
réutilisé)

A l'inverse, si on n'utilise pas les pointeurs, de
nombreuses copies sont faites (et possiblement
profondeurs)

Pour faire le lien avec UML,
c'est une question indirectement soulevée ici :



Etude de cas 2 :

copies liées à l'utilisation de vector

avec vector ?

```
class Point {
public:
    Point(int x=0,int y=0);

    virtual ~Point();
    int abs, ord;
};

ostream & operator<<(ostream &, const Point &);
```

```
Point::Point(int x, int y):abs{x}, ord{y} {
    cout << "creation de " << *this << endl;
}
```

```
Point::~~Point() {
    cout << "destruction de " << *this << endl;
}
```

```
ostream & operator<<(ostream & out, const Point & p) {
    out << "(" << p.abs << ", " << p.ord << ") ";
    return out;
}
```

```
int main() {
    Point p0;
    vector<Point> v;
    v.push_back(p0);
    return EXIT_SUCCESS;
}
```

?

avec vector ?

```
class Point {  
public:  
    Point(int x=0,int y=0);  
  
    virtual ~Point();  
    int abs, ord;  
};  
ostream & operator<<(ostream &, const Point &);
```

```
int main() {  
    Point p0;  
    vector<Point> v;  
    v.push_back(p0);  
    return EXIT_SUCCESS;  
}
```

```
Point::Point(int x, int y):abs{x}, ord{y} {  
    cout << "creation de " << *this << endl;  
}  
  
Point::~~Point() {  
    cout << "destruction de " << *this << endl;  
}  
ostream & operator<<(ostream & out, const Point & p) {  
    out << "(" << p.abs << ", " << p.ord << ")";  
    return out;  
}
```

```
creation de (0,0)  
destruction de (0,0)  
destruction de (0,0)
```

2 destructions, c'est donc qu'il y a copie

avec vector ?

```
class Point {
public:
    Point(int x=0,int y=0);
    Point(const Point& orig);
    virtual ~Point();
    int abs, ord;
};
ostream & operator<<(ostream &, const Point &);
```

```
int main() {
    Point p0;
    vector<Point> v;
    v.push_back(p0);
    return EXIT_SUCCESS;
}
```

```
Point::Point(int x, int y):abs{x}, ord{y} {
    cout << "creation de " << *this << endl;
}
Point::Point(const Point& orig): abs{orig.abs},ord{orig.ord} {
    cout << "copie de " << orig << endl;
}
Point::~~Point() {
    cout << "destruction de " << *this << endl;
}
ostream & operator<<(ostream & out, const Point & p) {
    out << "(" << p.abs << "," << p.ord<< ") ";
    return out;
}
```

```
creation de (0,0)
copie de (0,0)
destruction de (0,0)
destruction de (0,0)
```

oui, vector stocke des copies

Il faut réfléchir à ce que l'on veut ...

```
int main() {  
    Point p0;  
    vector<Point> v;  
    v.push_back(p0);  
    return EXIT_SUCCESS;  
}
```

```
creation de (0,0)  
copie de (0,0)  
destruction de (0,0)  
destruction de (0,0)
```

```
int main() {  
    vector<Point> v;  
    v.emplace_back(0,0);  
    return EXIT_SUCCESS;  
}
```

```
creation de (0,0)  
destruction de (0,0)
```

```
int main() {  
    Point *p0{0,0};  
    vector<Point*> v;  
    v.push_back(p0);  
    delete p0; // ?!  
    return EXIT_SUCCESS;  
}
```

```
creation de (0,0)  
destruction de (0,0)
```

Il faut réfléchir à ce que l'on veut ...

```
int main() {  
    Point p0;  
    vector<Point> v;  
    v.push_back(p0);  
    return EXIT_SUCCESS;  
}
```

```
creation de (0,0)  
copie de (0,0)  
destruction de (0,0)  
destruction de (0,0)
```

```
int main() {  
    vector<Point> v;  
    v.emplace_back(0,0);  
    return EXIT_SUCCESS;  
}
```

```
creation de (0,0)  
destruction de (0,0)
```

```
int main() {  
    Point *p0{0,0};  
    vector<Point*> v;  
    v.push_back(p0);  
    v.clear(); //+prudent  
    delete p0;  
    return EXIT_SUCCESS;  
}
```

```
creation de (0,0)  
destruction de (0,0)
```

Etude de cas 3 :
copies dans un exemple plus long

```
class Polygone {
public:
    Polygone(initializer_list<Point> );
    vector<Point> allPoints;
};
ostream & operator<<(ostream &, const Polygone &);
```

```
Polygone::Polygone(initializer_list<Point> l):allPoints{l} {}

ostream & operator<<(ostream &out, const Polygone &q) {
    for (Point p:q.allPoints) out << p << endl;
    return out;
}
```

```
int main() {
    Point p0, p1{1,1}, p2{2,2}, p3{3,3};
    Polygone segm1{p0,p1};
    Polygone segm2{p2,p3};
    vector<Polygone> vp {segm1, segm2};
    for (Polygone q:vp) cout << q ;
    return EXIT_SUCCESS;
}
```

donnez une estimation du nb copie/destruction ...

```

class Polygone {
public:
    Polygone(initializer_list<Point> );
    vector<Point> allPoints;
};
ostream & operator<<(ostream &, const Polygone &);

```

```

Polygone::Polygone(initializer_list<Point> l):allPoints{l} {}

ostream & operator<<(ostream &out, const Polygone &q) {
    for (Point p:q.allPoints) out << p << endl;
    return out;
}

```

```

int main() {
    Point p0, p1{1,1}, p2{2,2}, p3{3,3};
    Polygone segm1{p0,p1};
    Polygone segm2{p2,p3};
    vector<Polygone> vp {segm1, segm2};
    for (Polygone q:vp) cout << q ;
    return EXIT_SUCCESS;
}

```

4 créations de points + 24 copies
28 destructions

```

creation de (0,0)
creation de (1,1)
creation de (2,2)
creation de (3,3)
copie de (0,0)
copie de (1,1)
copie de (0,0)
copie de (1,1)
destruction de (1,1)
destruction de (0,0)
copie de (2,2)
copie de (3,3)
copie de (2,2)
copie de (3,3)
destruction de (3,3)
destruction de (2,2)
copie de (0,0)
copie de (1,1)
copie de (2,2)
copie de (3,3)
copie de (0,0)
copie de (1,1)
copie de (2,2)
copie de (3,3)
destruction de (2,2)
destruction de (3,3)
destruction de (0,0)
destruction de (1,1)
copie de (0,0)
copie de (1,1)
copie de (0,0)
(0,0)
destruction de (0,0)
copie de (1,1)
(1,1)
destruction de (1,1)
destruction de (0,0)
destruction de (1,1)
copie de (2,2)
copie de (3,3)
copie de (2,2)
(2,2)
destruction de (2,2)
copie de (3,3)
(3,3)
destruction de (3,3)
destruction de (2,2)
destruction de (3,3)
destruction de (0,0)
destruction de (1,1)
destruction de (2,2)
destruction de (3,3)
destruction de (2,2)
destruction de (3,3)
destruction de (0,0)
destruction de (1,1)
destruction de (3,3)
destruction de (2,2)
destruction de (1,1)
destruction de (0,0)

```



```

class Polygone {
public:
    Polygone(initializer_list<Point> );
    vector<Point> allPoints;
};
ostream & operator<<(ostream &, const Polygone &);

```

```

Polygone::Polygone(initializer_list<Point> l):allPoints{l} {}

ostream & operator<<(ostream &out, const Polygone &q) {
    for (Point p:q.allPoints) out << p << endl;
    return out;
}

```

```

int main() {
    Point p0, p1{1,1}, p2{2,2}, p3{3,3};
    Polygone segm1{p0,p1};
    Polygone segm2{p2,p3};
    vector<Polygone> vp {segm1, segm2};
    for (Polygone q:vp) cout << q ;
    return EXIT_SUCCESS;
}

```

```

creation de (0,0)
creation de (1,1)
creation de (2,2)
creation de (3,3)
copie de (0,0)
copie de (1,1)
copie de (0,0)
copie de (1,1)
destruction de (1,1)
destruction de (0,0)
copie de (2,2)
copie de (3,3)
copie de (2,2)
copie de (3,3)
destruction de (3,3)
destruction de (2,2)
copie de (0,0)
copie de (1,1)
copie de (2,2)
copie de (3,3)
copie de (0,0)
copie de (1,1)
copie de (2,2)
copie de (3,3)
destruction de (2,2)
destruction de (3,3)
destruction de (0,0)
destruction de (1,1)
copie de (0,0)
copie de (1,1)
copie de (0,0)
(0,0)
destruction de (0,0)
copie de (1,1)
(1,1)
destruction de (1,1)
destruction de (0,0)
destruction de (1,1)
copie de (2,2)
copie de (3,3)
copie de (2,2)
(2,2)
destruction de (2,2)
copie de (3,3)
(3,3)
destruction de (3,3)
destruction de (2,2)
destruction de (3,3)
destruction de (0,0)
destruction de (1,1)
destruction de (2,2)
destruction de (3,3)
destruction de (2,2)
destruction de (3,3)
destruction de (0,0)
destruction de (1,1)
destruction de (3,3)
destruction de (2,2)
destruction de (1,1)
destruction de (0,0)

```

```
class Polygone {
public:
    Polygone(initializer_list<Point> );
    vector<Point> allPoints;
};
ostream & operator<<(ostream &, const Polygone &);
```

```
Polygone::Polygone(initializer_list<Point> l):allPoints{l} {}

ostream & operator<<(ostream &out, const Polygone &q) {
    for (Point &p:q.allPoints) out << p << endl;
    return out;
}
```

```
int main() {
    Point p0, p1{1,1}, p2{2,2}, p3{3,3};
    Polygone segm1{p0,p1};
    Polygone segm2{p2,p3};
    vector<Polygone> vp {segm1, segm2};
    for (Polygone &q:vp) cout << q ;
    return EXIT_SUCCESS;
}
```

utilisons une référence dans les for pour éviter qqs copies
elle pose pb dans le 1er cas, voyez vous pourquoi ?

```
class Polygone {
public:
    Polygone(initializer_list<Point> );
    vector<Point> allPoints;
};
ostream & operator<<(ostream &, const Polygone &);
```

```
Polygone::Polygone(initializer_list<Point> l):allPoints{l} {}

ostream & operator<<(ostream &out, const Polygone &q) {
    for (Point &p:q.allPoints) out << p << endl;
    return out;
}
```

```
int main() {
    Point p0, p1{1,1}, p2{2,2}, p3{3,3};
    Polygone segm1{p0,p1};
    Polygone segm2{p2,p3};
    vector<Polygone> vp {segm1, segm2};
    for (Polygone &q:vp) cout << q ;
    return EXIT_SUCCESS;
}
```

le const de l'argument exprime que les constituants doivent être invariants -> il faut le dire dans le type de p

```
class Polygone {
public:
    Polygone(initializer_list<Point> );
    vector<Point> allPoints;
};
ostream & operator<<(ostream &, const Polygone &);
```

```
Polygone::Polygone(initializer_list<Point> l):allPoints{l} {}

ostream & operator<<(ostream &out, const Polygone &q) {
    for (Point const &p:q.allPoints) out << p << endl;
    return out;
}
```

```
int main() {
    Point p0, p1{1,1}, p2{2,2}, p3{3,3};
    Polygone segm1{p0,p1};
    Polygone segm2{p2,p3};
    vector<Polygone> vp {segm1, segm2};
    for (Polygone &q:vp) cout << q ;
    return EXIT_SUCCESS;
}
```

```

class Polygone {
public:
    Polygone(initializer_list<Point> );
    vector<Point> allPoints;
};
ostream & operator<<(ostream &, const Polygone &);

```

```

Polygone::Polygone(initializer_list<Point> l):allPoints{l} {}

ostream & operator<<(ostream &out, const Polygone &q) {
    for (Point const &p:q.allPoints) out << p << endl;
    return out;
}

```

```

int main() {
    Point p0, p1{1,1}, p2{2,2}, p3{3,3};
    Polygone segm1{p0,p1};
    Polygone segm2{p2,p3};
    vector<Polygone> vp {segm1, segm2};
    for (Polygone &q:vp) cout << q ;
    return EXIT_SUCCESS;
}

```

4 créations de points + 16 copies
(au lieu de 4+24)

```

creation de (0,0)
creation de (1,1)
creation de (2,2)
creation de (3,3)
copie de (0,0)
copie de (1,1)
copie de (0,0)
copie de (1,1)
destruction de (1,1)
destruction de (0,0)
copie de (2,2)
copie de (3,3)
copie de (2,2)
copie de (3,3)
destruction de (3,3)
destruction de (2,2)
copie de (0,0)
copie de (1,1)
copie de (2,2)
copie de (3,3)
copie de (0,0)
copie de (1,1)
copie de (2,2)
copie de (3,3)
destruction de (2,2)
destruction de (3,3)
destruction de (0,0)
destruction de (1,1)
(0,0)
(1,1)
(2,2)
(3,3)
destruction de (0,0)
destruction de (1,1)
destruction de (2,2)
destruction de (3,3)
destruction de (2,2)
destruction de (3,3)
destruction de (0,0)
destruction de (1,1)
destruction de (3,3)
destruction de (2,2)
destruction de (1,1)
destruction de (0,0)

```

```
class Polygone {
public:
    Polygone(initializer_list<Point> );
    vector<Point> allPoints;
};
ostream & operator<<(ostream &, const Polygone &);
```

```
Polygone::Polygone(initializer_list<Point> l):allPoints{l} {}

ostream & operator<<(ostream &out, const Polygone &q) {
    for (Point const &p:q.allPoints) out << p << endl;
    return out;
}
```

```
int main() {
    Point p0, p1{1,1}, p2{2,2}, p3{3,3};
    Polygone segm1{p0,p1};
    Polygone segm2{p2,p3};
    vector<Polygone> vp {segm1, segm2};
    for (Polygone &q:vp) cout << q ;
    return EXIT_SUCCESS;
}
```

creation de (0,0)
 creation de (1,1)
 creation de (2,2)
 creation de (3,3)
 copie de (0,0)
 copie de (1,1)
 copie de (0,0)
 copie de (1,1)
 destruction de (1,1)
 destruction de (0,0)
 copie de (2,2)
 copie de (3,3)
 copie de (2,2)
 copie de (3,3)
 destruction de (3,3)
 destruction de (2,2)
 copie de (0,0)
 copie de (1,1)
 copie de (2,2)
 copie de (3,3)
 copie de (0,0)
 copie de (1,1)
 copie de (2,2)
 copie de (3,3)
 destruction de (2,2)
 destruction de (3,3)
 destruction de (0,0)
 destruction de (1,1)
 (0,0)
 (1,1)
 (2,2)
 (3,3)
 destruction de (0,0)
 destruction de (1,1)
 destruction de (2,2)
 destruction de (3,3)
 destruction de (2,2)
 destruction de (3,3)
 destruction de (0,0)
 destruction de (1,1)
 destruction de (3,3)
 destruction de (2,2)
 destruction de (1,1)
 destruction de (0,0)

faire mieux ?

l'écriture des listes d'initialisations engendre une copie intermédiaire inutile,
 en plus celles qui sont naturelles à vector

```
class Polygone {
public:
    Polygone(initializer_list<Point> );
    vector<Point> allPoints;
};
ostream & operator<<(ostream &, const Polygone &);
```

```
Polygone::Polygone(initializer_list<Point> l):allPoints{l} {}

ostream & operator<<(ostream &out, const Polygone &q) {
    for (Point const &p:q.allPoints) out << p << endl;
    return out;
}
```

```
int main() {
    Point p0, p1{1,1}, p2{2,2}, p3{3,3};
    Polygone segm1{p0,p1};
    Polygone segm2{p2,p3};
    vector<Polygone> vp {segm1, segm2};
    for (Polygone &q:vp) cout << q ;
    return EXIT_SUCCESS;
}
```

creation de (0,0)
 creation de (1,1)
 creation de (2,2)
 creation de (3,3)
 copie de (0,0)
 copie de (1,1)
 copie de (0,0)
 copie de (1,1)
 destruction de (1,1)
 destruction de (0,0)
 copie de (2,2)
 copie de (3,3)
 copie de (2,2)
 copie de (3,3)
 destruction de (3,3)
 destruction de (2,2)
 copie de (0,0)
 copie de (1,1)
 copie de (2,2)
 copie de (3,3)
 copie de (0,0)
 copie de (1,1)
 copie de (2,2)
 copie de (3,3)
 destruction de (2,2)
 destruction de (3,3)
 destruction de (0,0)
 destruction de (1,1)
 (0,0)
 (1,1)
 (2,2)
 (3,3)
 destruction de (0,0)
 destruction de (1,1)
 destruction de (2,2)
 destruction de (3,3)
 destruction de (2,2)
 destruction de (3,3)
 destruction de (0,0)
 destruction de (1,1)
 destruction de (3,3)
 destruction de (2,2)
 destruction de (1,1)
 destruction de (0,0)

ces copies sont intrinsèques à l'utilisation des listes d'initialisations
 (discussions hors programme c++ initializer_list always copy)

```
class Polygone {  
public:  
    Polygone(initializer_list<Point> );  
    vector<Point> allPoints;  
};
```

```
Polygone::Polygone(initializer_list<Point> l):allPoints{l} {}
```

```
int main() {  
    Point p0, p1{1,1}, p2{2,2}, p3{3,3};  
    Polygone segm1{p0,p1};  
    ...  
}
```

on remplace les listes d'initialisation pour éviter des copies


```
class Polygone {  
public:  
    Polygone(initializer_list<Point> );  
    vector<Point> allPoints;  
};
```

```
Polygone::Polygone(initializer_list<Point> l):allPoints{l} {}
```

```
int main() {  
    Point p0, p1{1,1}, p2{2,2}, p3{3,3};  
    Polygone segm1{};  
    segm1.add(p0).add(p1); // à écrire  
    ...  
}
```

```
class Polygone {  
public:  
    Polygone(initializer_list<Point> );  
    vector<Point> allPoints;  
    Polygone& add(Point &);  
};
```

```
Polygone::Polygone(initializer_list<Point> l):allPoints{l} {}  
Polygone& Polygone::add(Point &x){ allPoints.push_back(x); return *this;}
```

```
int main() {  
    Point p0, p1{1,1}, p2{2,2}, p3{3,3};  
    Polygone segm1{};  
    segm1.add(p0).add(p1);  
    ...  
}
```

```
class Polygone {
public:
    Polygone(initializer_list<Point> );
    vector<Point> allPoints;
    Polygone& add(Point &);
};
```

```
Polygone::Polygone(initializer_list<Point> l):allPoints{l} {}
Polygone& Polygone::add(Point &x){ allPoints.push_back(x); return *this;}
```

```
int main() {
    Point p0, p1{1,1}, p2{2,2}, p3{3,3};
    Polygone segm1{};
    segm1.add(p0).add(p1);
    ...
}
```

```
... création des 4 points
... effet de l'ajout à segm1 :
copie de (0,0)
copie de (1,1)
copie de (0,0)
destruction de (0,0)
... suite du programme
```

on obtient cela ... qui ne sont que conséquence des ajouts au vector ...?!

```
class Polygone {
public:
    Polygone(initializer_list<Point> );
    vector<Point> allPoints;
    Polygone& add(Point &);
};
```

```
Polygone::Polygone(initializer_list<Point> l):allPoints{l} {}
Polygone& Polygone::add(Point &x){ allPoints.push_back(x); return *this;}
```

```
int main() {
    Point p0, p1{1,1}, p2{2,2}, p3{3,3};
    Polygone segm1{};
    segm1.add(p0).add(p1);
    ...
}
```

```
... création des 4 points
... effet de l'ajout à segm1 :
copie de (0,0)
copie de (1,1)
copie de (0,0)
destruction de (0,0)
... suite du programme
```

on comprend alors que la capacité du vector est dépassée au second ajout
Le premier vector contenant p0 est donc recopié dans un vector +grand
Ce qui explique la nouvelle copie de p0, et la destruction de l'ancien p0

```
class Polygone {
public:
    ...
    Polygone(int );
    ...
};
ostream & operator<<(ostream &, const Polygone &);
```

```
Polygone::Polygone(int x){allPoints.reserve(x);} // reserve fixe la capacité
// Rq : la liste d'initialisation est vide car le constructeur par défaut de
vecteur convient
```

```
int main() {
    Point p0, p1{1,1}, p2{2,2}, p3{3,3};
    Polygone segm1(2);
    segm1.add(p0).add(p1);
    ...
}
```

```
... création des 4 points
... effet de l'ajout à segm1 :
copie de (0,0)
copie de (1,1)

... suite du programme
```

on peut fixer une capacité initiale suffisante
(et faire idem pour la suite du main)

```

int main() {
    Point p0, p1{1,1}, p2{2,2}, p3{3,3};
    Polygone segm1(2);
    segm1.add(p0).add(p1);

    Polygone segm2(2);
    segm2.add(p2).add(p3);

    vector<Polygone> vp;
    vp.reserve(2);
    vp.push_back(segm1);
    vp.push_back(segm2);

    for (Polygone &q:vp) cout << q ;
    return EXIT_SUCCESS;
}

```

```

creation de (0,0)
creation de (1,1)
creation de (2,2)
creation de (3,3)

```

```

copie de (0,0)
copie de (1,1)
copie de (2,2)
copie de (3,3)
copie de (0,0)
copie de (1,1)
copie de (2,2)
copie de (3,3)

```

```

(0,0)
(1,1)
(2,2)
(3,3)

```

```

destruction de (0,0)
destruction de (1,1)
destruction de (2,2)
destruction de (3,3)
destruction de (2,2)
destruction de (3,3)
destruction de (0,0)
destruction de (1,1)
destruction de (3,3)
destruction de (2,2)
destruction de (1,1)
destruction de (0,0)

```

4 créations de points + 8 copies
(au lieu de 4+24) - Conforme à ce qu'on voulait

Etude de cas 3, conclusion :

de nombreuses copies sont apparues de manière inattendues :

- dans le parcours à l'affichage
- dans les listes d'initialisation
- dans les dépassements de capacité de vector

Les copies pour vector, elles, étaient attendues, elles correspondent au choix de conception. (Sinon on aurait utilisé un vecteur de pointeurs vers des Points)

Nous avons fait 3 petites études de cas
et discuté pas mal de copies/destructions

Cas 1 : destructions problématiques

Cas 2 : copies liées à vector

Cas 3 : copies dans un exemple plus long

Copies/destructions ne sont pas anecdotiques :
il vous faut comprendre quelles sont les coûts à
l'exécution du code que vous écrivez.

il nous reste encore un morceau :

L'affectation

Si on a :

```
class A{  
  B *my_b;  
  public : A();  
}
```

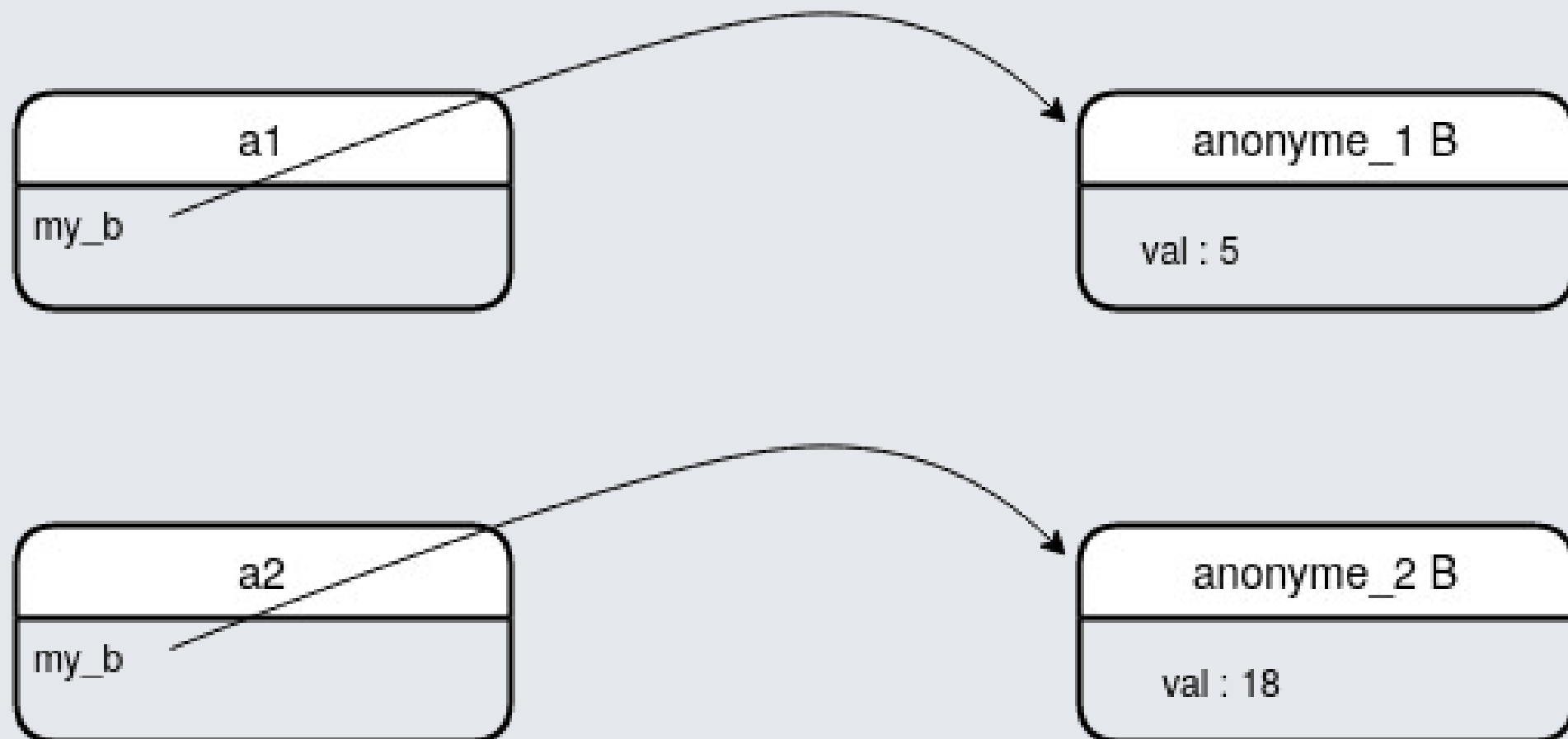
```
A::A() : my_b{new B()} {}
```

```
class B{  
  int val;  
  public : B();  
}
```

```
B::B() : val{rand()} {}
```

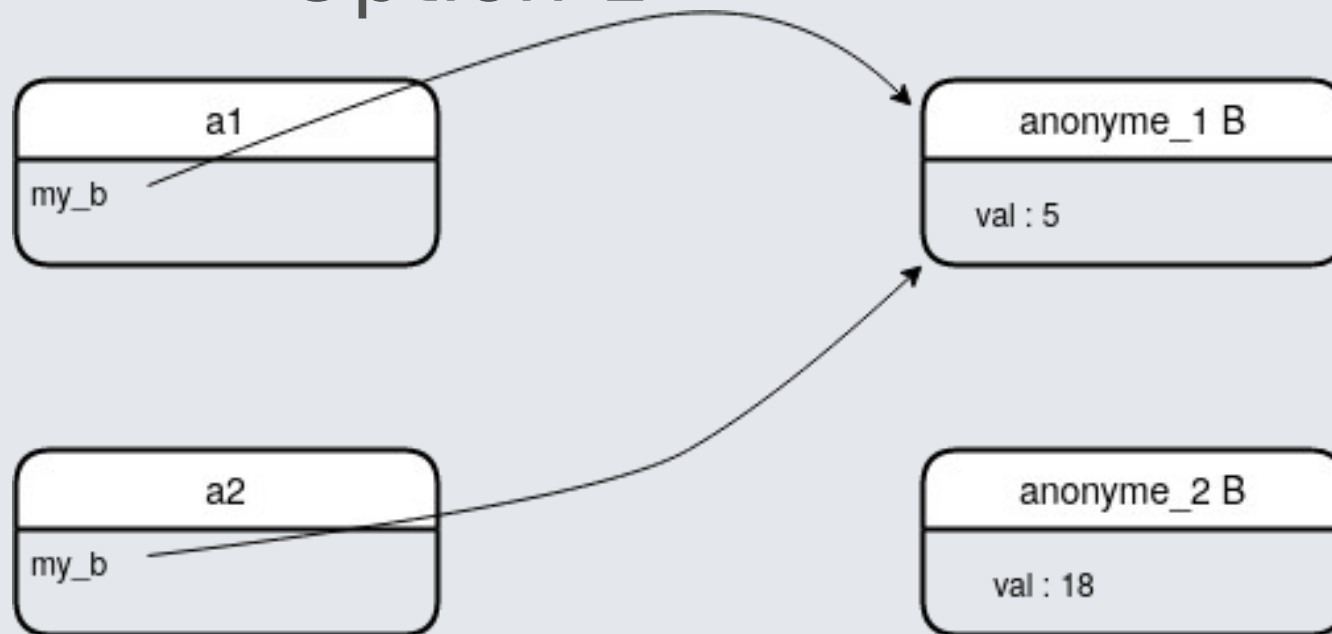
```
int main() {  
  A a1, a2;  
  a2 = a1;  
}
```

Quel effet attend-on ?

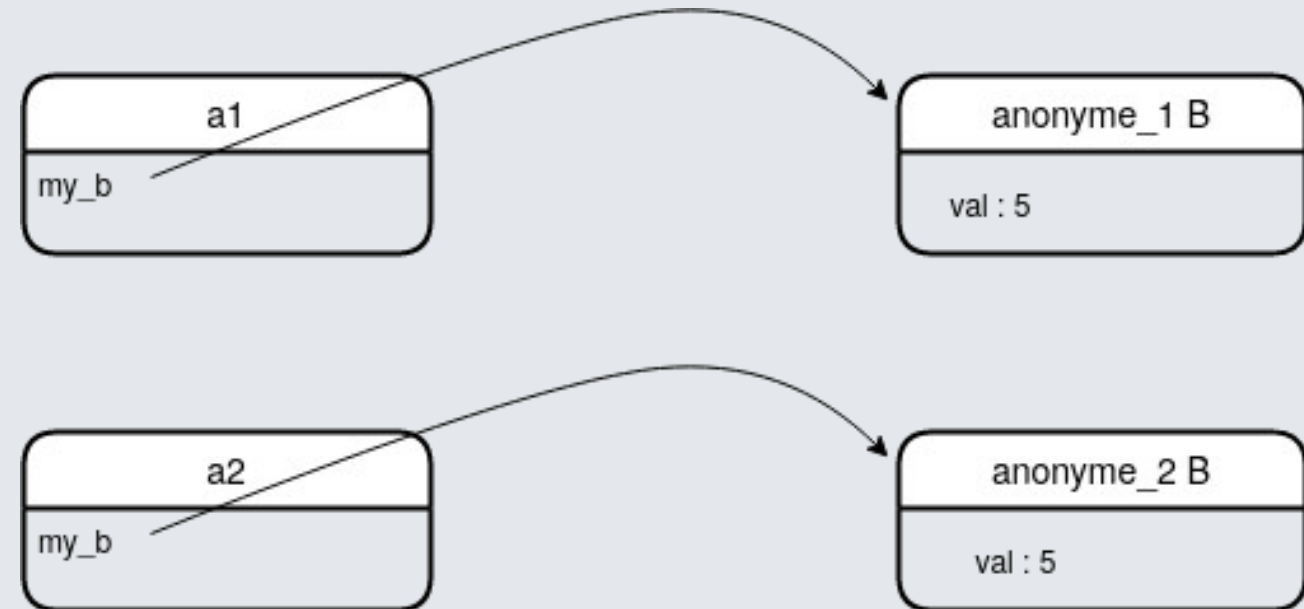


Plusieurs possibilités ...

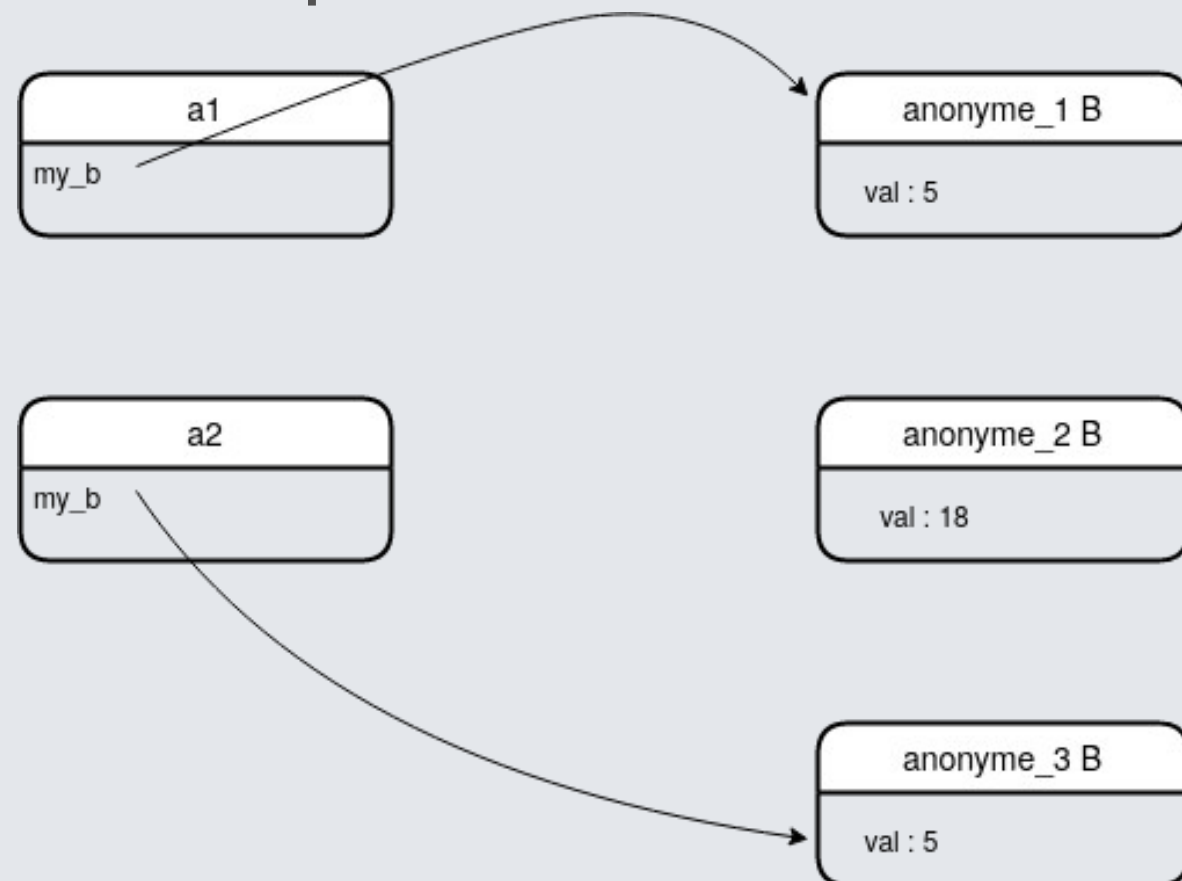
Option 1



Option 2



Option 3



C'est un problème proche de celui de la construction d'une copie.

Avec en plus celui de la gestion des anciens constituants de a2

- Ne serait-ce que pour clarifier : il est nécessaire de redéfinir l'opérateur d'affectation.
- L'affectation présente par défaut fait l'affectation membre à membre etc ... Dans notre illustration , il s'agit du cas 1, **sans rendre au système la mémoire d'anonyme_2**, on ne peut donc pas le tolérer tel quel...

Signature habituelle de sa déclaration :

```
class A {  
    const A & operator=(const A &x) ;  
};
```

Quelques remarques :

- l'affectation est vue comme une opération membre (c'est syntaxiquement contraint par c++)
- `a1=a2` est strictement équivalent à `a1.operator=(a2)`
c'est `a1` qui réalise l'affectation à partir de l'argument `a2`
- on peut assez naturellement considérer `x` constant
- le passage par référence permet d'éviter une copie locale de `a2`
- la nature du type retour est liée à des usages de programmeur (voir slide suivant)
- la visibilité `public/private`, le type retour et du paramètre sont très librement modifiables ...

Signature habituelle de sa déclaration :

```
class A {  
    const A & operator=(const A &x) ;  
};
```

la nature du type retour est liée à des usages de programmeur ...

Probablement que vous ne l'avez jamais essayé, mais ceci compile :

```
int a{0},b{1},c{2};  
a=b=c;  
cout << a << b << c << endl;
```

Signature habituelle de sa déclaration :

```
class A {  
    const A & operator=(const A &x) ;  
};
```

la nature du type retour est liée à des usages de programmeur ...

Probablement que vous ne l'avez jamais essayé, mais ceci compile :

```
int a{0},b{1},c{2};  
a=b=c;  
cout << a << b << c << endl;
```

Signature habituelle de sa déclaration :

```
class A {  
    const A & operator=(const A &x) ;  
};
```

la nature du type retour est liée à des usages de programmeur ...

Probablement que vous ne l'avez jamais essayé, mais ceci compile :

```
int a{0},b{1},c{2};  
a=(b=c) ;  
cout << a << b << c << endl;
```

(écriture équivalente)

Signature habituelle de sa déclaration :

```
class A {  
    const A & operator=(const A &x) ;  
};
```

la nature du type retour est liée à des usages de programmeur ...

Probablement que vous ne l'avez jamais essayé, mais ceci compile :

```
int a{0},b{1},c{2};  
a=(b=c) ;  
cout << a << b << c << endl;
```

(écriture équivalente)

c.a.d : associativité à droite
et "propagation" des affectations

Signature habituelle de sa déclaration :

```
class A {  
    const A & operator=(const A &x) ;  
};
```

la nature du type retour est liée à des usages de programmeur ...

On a donc tendance à généraliser pour permettre :

```
A a, b, c ;  
a = (b = c) ;
```

L'idée est de "reprendre" le terme le plus à droite au fur et à mesure de la résolution de l'associativité.

C'est pourquoi le type retour suggéré est celui de l'argument (avec une référence pour éviter une copie)

Signature habituelle de sa déclaration :

```
class A {  
    const A & operator=(const A &x) ;  
};
```

la visibilité public/private, le type retour et du paramètre sont très librement modifiables ...

Ce sont des choses qu'on a déjà illustré sur le constructeur de copie :

- le déclarer private complique les usages de A
- déclarer l'argument non const peut permettre un "move-assignement"
- que le type retour soit const ou pas n'a pas de grande importance : dans `a=b=c` un non const peut être considéré const
- si pour le type retour on choisi void, on interdit de fait l'écriture `a=b=c`;

Signature habituelle de sa déclaration :

```
class A {  
    const A & operator=(const A &x) ;  
};
```

la visibilité public/private, le type retour et du paramètre sont très librement modifiables ...

On pourrait éventuellement imaginer écrire

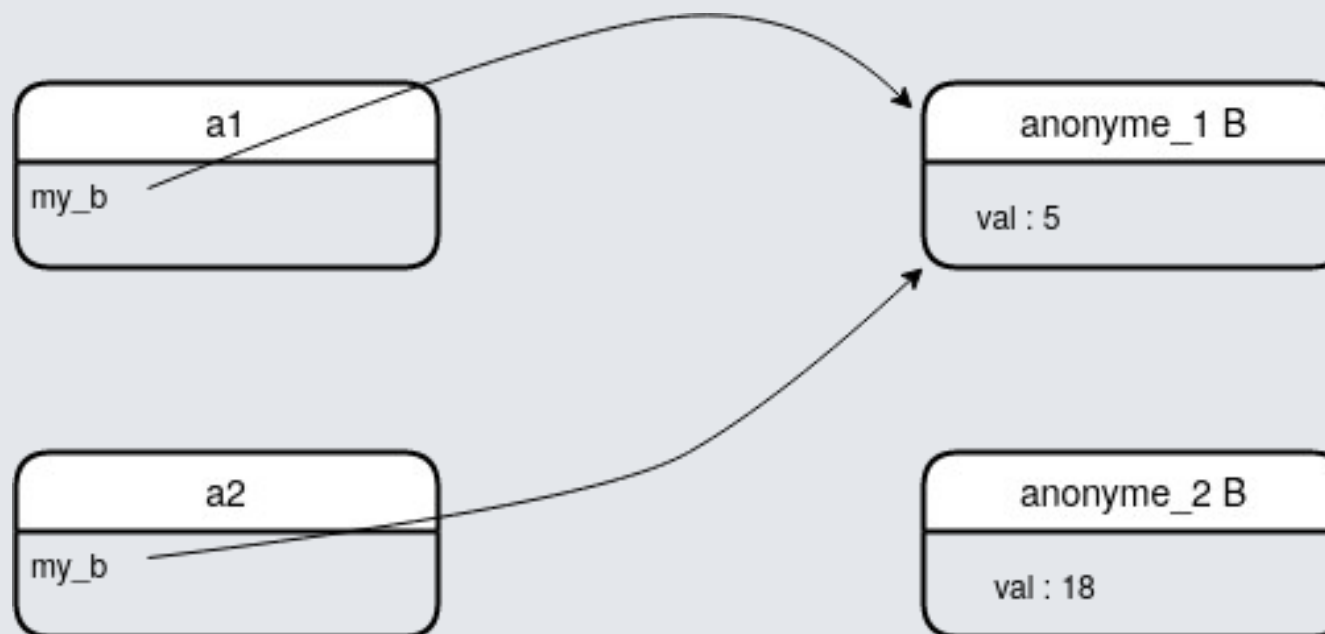
```
class A {  
    A & operator=(A &x) const ;  
};
```

Ce const appliqué à la méthode signifiant que dans $a=b$ a serait inchangé ... bizarre, mais bon ...

Son implémentation (objectif option 1) :

```
const A & A::operator=(const A &x) {  
    . . .  
}
```

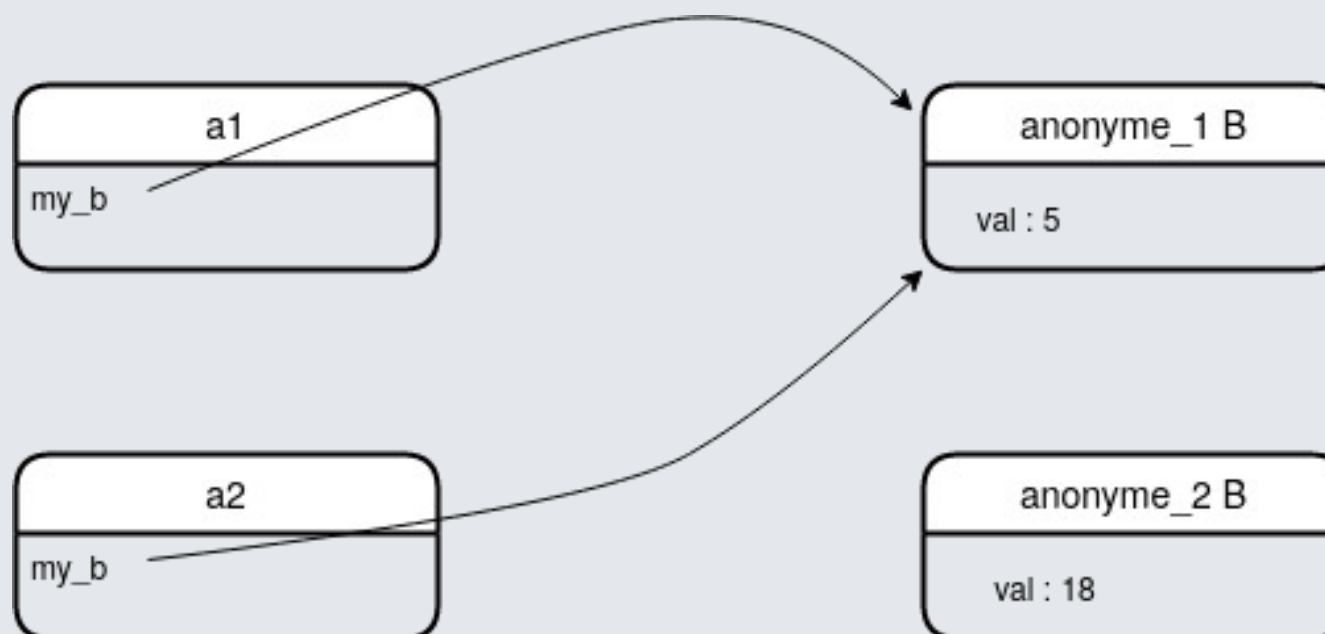
a2=a1;



Son implémentation (objectif option 1) :

```
const A & A::operator=(const A &x) {  
    delete(my_b); // libère anonyme_2  
    my_b = x.my_b; // =entre pointeurs  
    ... // retourne qq chose  
}
```

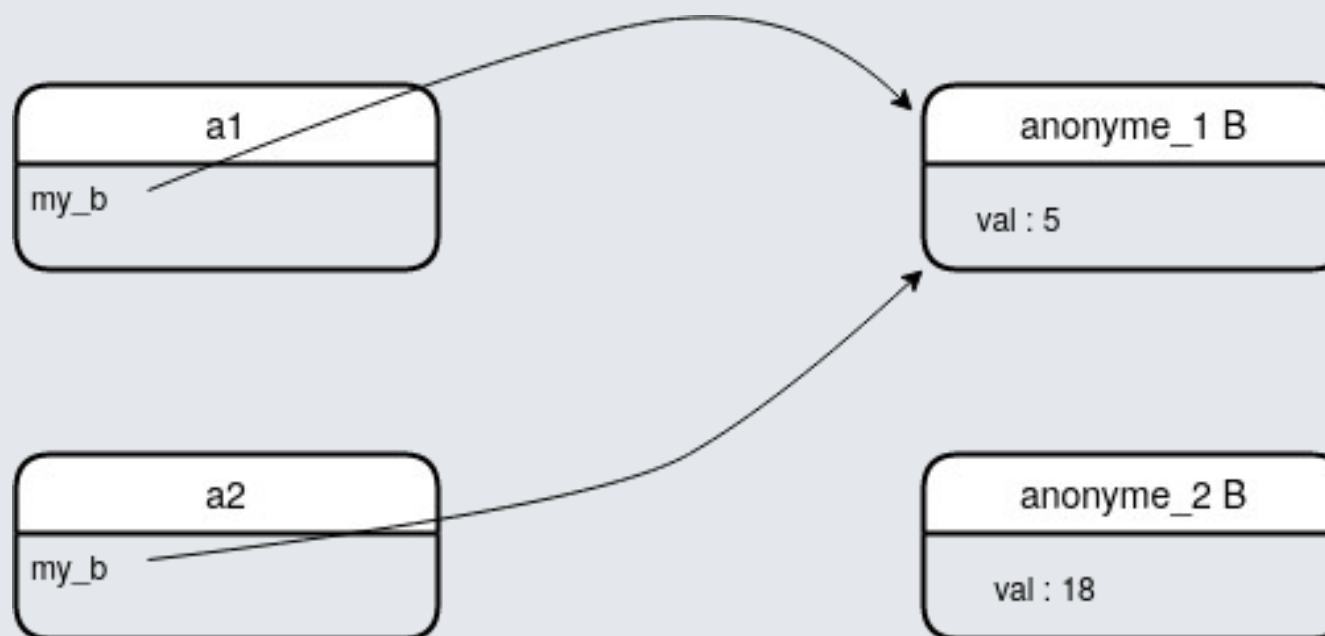
a2=a1;



Son implémentation (objectif option 1) :

```
const A & A::operator=(const A &x) {  
    delete(my_b); // libère anonyme_2  
    my_b = x.my_b; // =entre pointeurs  
    return x; // par exemple  
    ...  
}
```

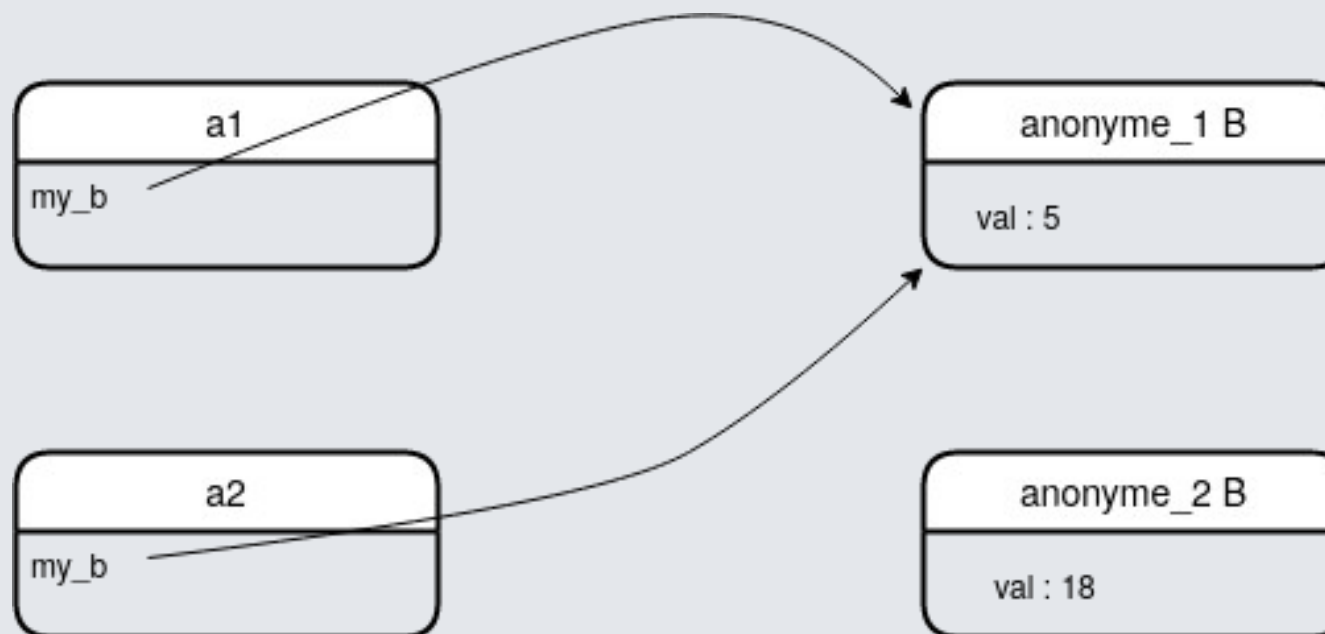
a2=a1;



Son implémentation (objectif option 1) :

```
const A & A::operator=(const A &x) {  
    delete(my_b); // libère anonyme_2  
    my_b = x.my_b; // =entre pointeurs  
    return *this; // aussi bien  
    ...  
}
```

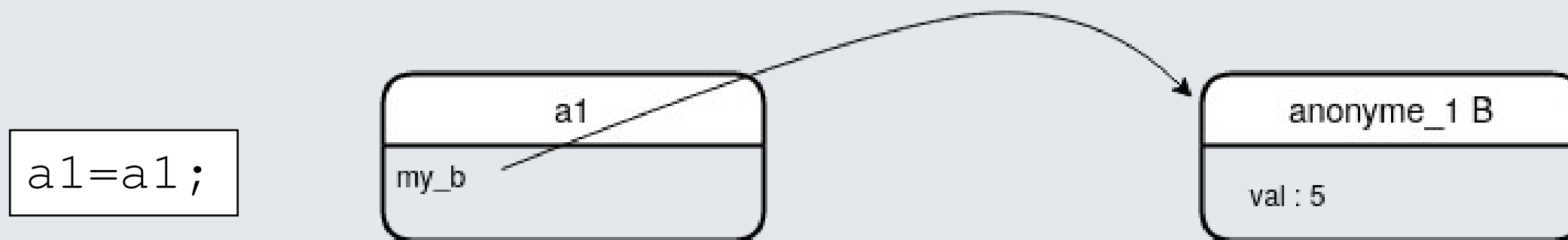
a2=a1;



Son implémentation (objectif option 1) :

```
const A & A::operator=(const A &x) {  
    delete(my_b); // libère anonyme_2  
    my_b = x.my_b; // =entre pointeurs  
    return *this; // aussi bien  
    ...  
}
```

Ce code est pas mal, mais encore imprudent ...
en effet, quid si le programmeur écrit :



Son implémentation (objectif option 1) :

```
const A & A::operator=(const A &x) {  
    if (*this==x) return *this;  
    delete(my_b);  
    my_b = x.my_b;  
    return *this;  
    . . .  
}
```

Son implémentation (objectif option 1) :

```
const A & A::operator=(const A &x) {  
    if (*this==x) return *this;  
    delete(my_b);  
    my_b = x.my_b;  
    return *this;  
    ...  
}
```

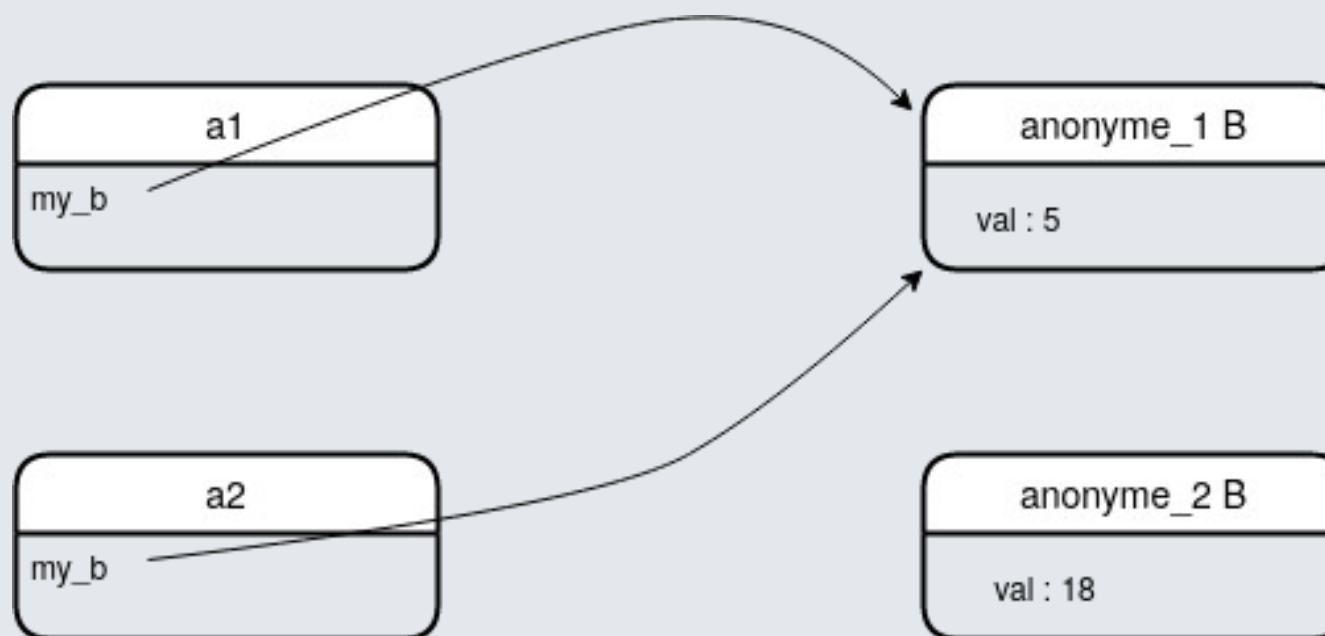
acceptable à 99,99 % ...

en effet == est aussi redéfinissable ...

autant utiliser == entre pointeurs plutôt qu'entre objets

Son implémentation (objectif option 1) :

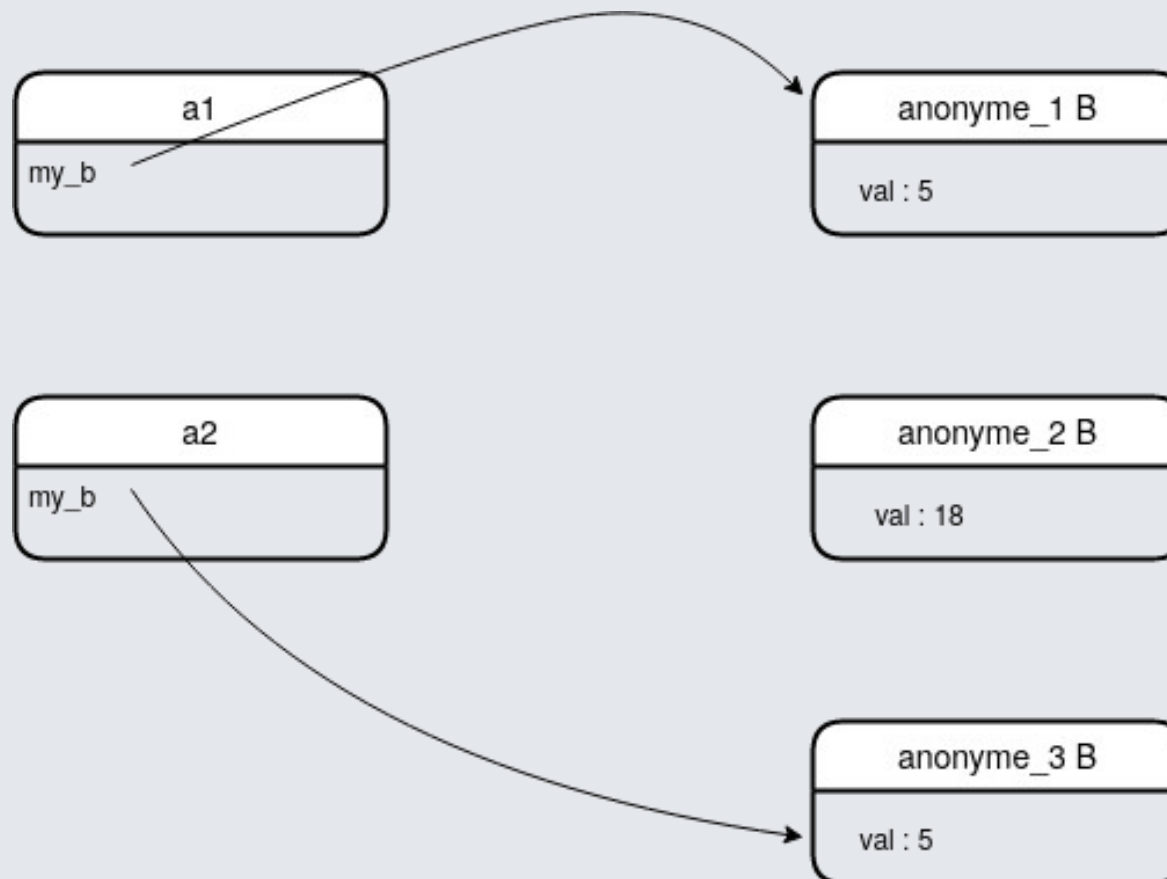
```
const A & A::operator=(const A &x) {  
    if (this == &x) return *this;  
    delete(my_b);  
    my_b = x.my_b;  
    return *this;  
}
```



Exercice - objectif option 3 :

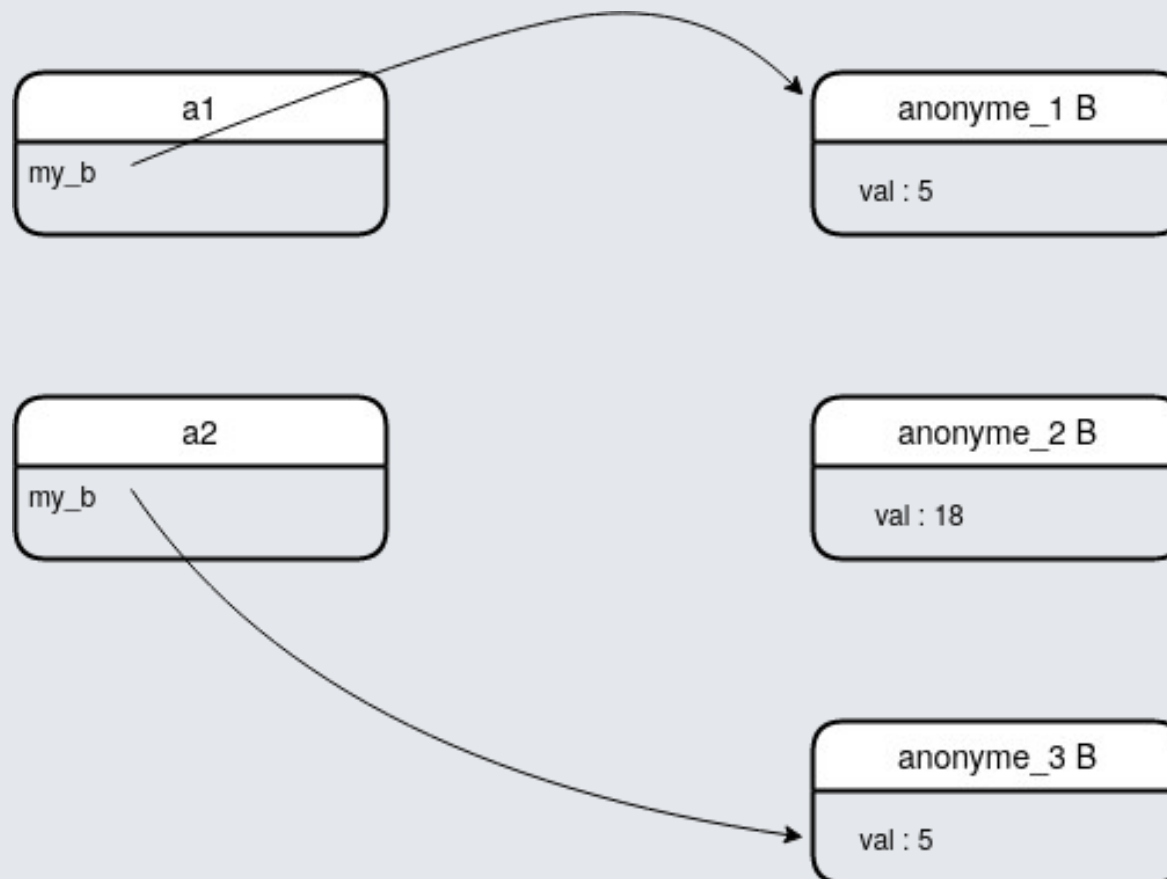
```
const A & A::operator=(const A &x) {  
    ...  
}
```

a2=a1;



Exercice - objectif option 3 :

```
const A & A::operator=(const A &x) {  
    if (this == &x) return *this;  
    delete(my_b);  
    my_b = new B{* (x.my_b)}; // copie par défaut..  
    return *this;  
}
```

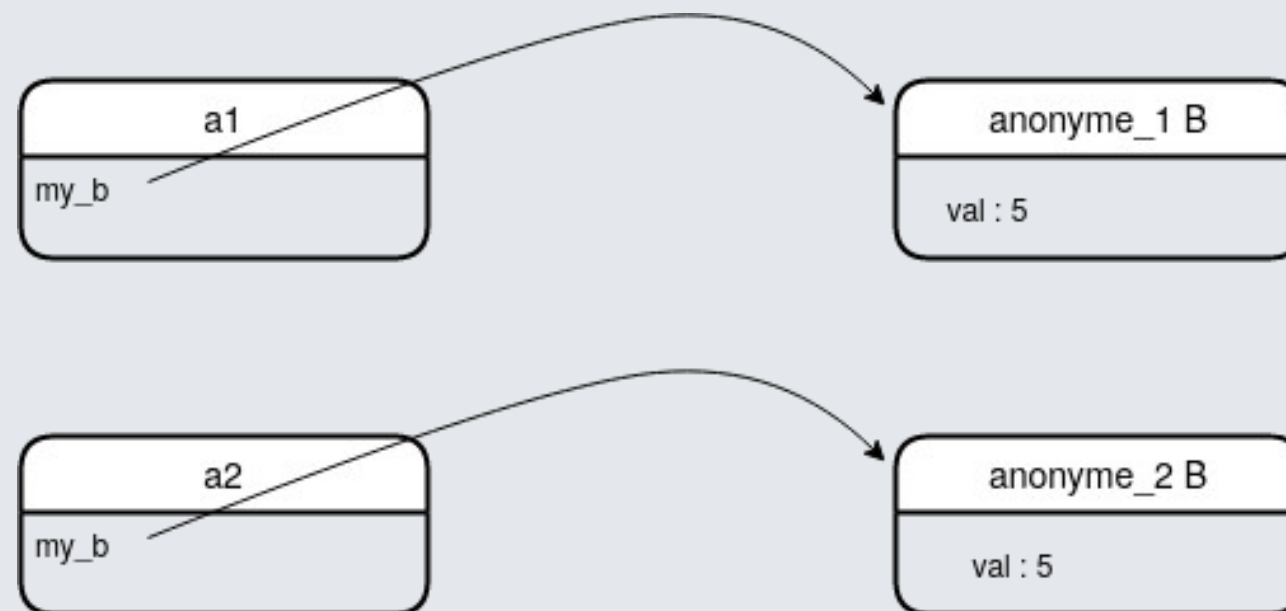


a2=a1;

Exercice - objectif option 2 :

```
const A & A::operator=(const A &x) {  
    ...  
}
```

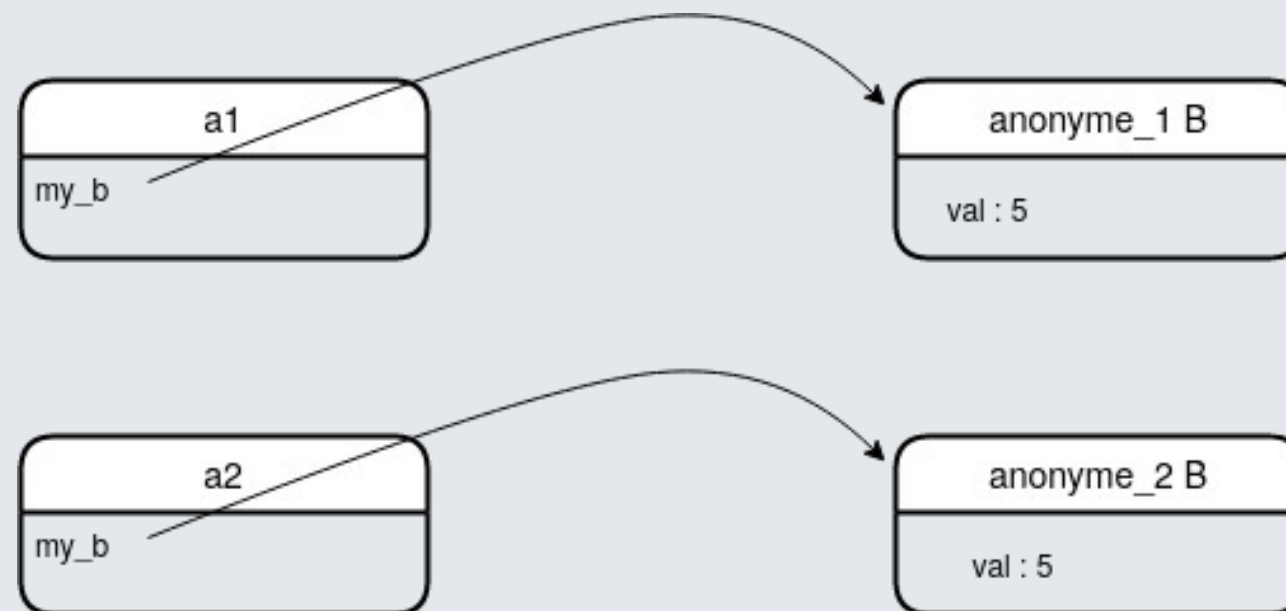
a2=a1;



Exercice - objectif option 2 :

```
const A & A::operator=(const A &x) {  
    *my_b = *(x.my_b); // affectation entre Bs ...  
    return *this;  
}
```

a2=a1;



Exercice - objectif option 2 :

```
const A & A::operator=(const A &x) {  
    *my_b = *(x.my_b); // affectation entre Bs ...  
    return *this;  
}
```

```
class A{  
    B *my_b;  
public : A();  
    const A & operator=(const A &x);  
}
```

```
class B{  
    int val;  
public : B();  
}
```

Mais on n'a pas redéfini cette affectation entre Bs !
Peut-on écrire ça ? ...

Exercice - objectif option 2 :

```
const A & A::operator=(const A &x) {  
    *my_b = *(x.my_b); // affectation entre Bs ...  
    return *this;  
}
```

```
class A{  
    B *my_b;  
public : A();  
    const A & operator=(const A &x);  
}
```

```
class B{  
    int val;  
public : B();  
}
```

Mais on n'a pas redéfini cette affectation entre Bs !
Peut-on écrire ça ? ...

oui, par défaut l'affectation est publique, et affecte membre à membre
c'est ce qu'on veut obtenir ici

Exercice - objectif option 2 :

```
const A & A::operator=(const A &x) {  
    *my_b = *(x.my_b); // affectation entre Bs ...  
    return *this;  
}
```

```
class A{  
    B *my_b;  
public : A();  
    const A & operator=(const A &x);  
}
```

```
class B{  
    int val;  
public : B();  
}
```

Mais val est private, c'est sûrement qu'on ne voulait pas pouvoir écrire :
my_b->val=(x.my_b)->val ... ?

Exercice - objectif option 2 :

```
const A & A::operator=(const A &x) {  
    *my_b = *(x.my_b); // affectation entre Bs ...  
    return *this;  
}
```

```
class A{  
    B *my_b;  
public : A();  
    const A & operator=(const A &x);  
}
```

```
class B{  
    int val;  
public : B();  
}
```

Mais val est private, c'est sûrement qu'on ne voulait pas pouvoir écrire :
my_b->val=(x.my_b)->val ... ?

On ne le fait pas directement. On passe bien par une méthode publique, tout est "légal". Elle donne implicitement des droits en modification !
Il faut donc faire attention !

Rule of three

Il vous faut penser aux :

Constructeurs par copie

Destructeurs

Opérateurs d'affectation

DÉFINITION CANONIQUE D'UNE CLASSE

(VOUS POUVEZ Y DÉROGER,
MAIS IL VOUS FAUT SAVOIR POURQUOI)

pour les tableaux

pour les copies

pour les
destructions

```
class ClasseCanon {  
public:  
    ClasseCanon();  
    ClasseCanon(const ClasseCanon &);  
    virtual ~ClasseCanon();  
    ClasseCanon &operator=(const ClasseCanon &);  
    friend ostream &  
        operator<<(ostream &, const ClasseCanon &);  
};
```

pour les affichages

pour les affectations

Rq : c++ définit une Rule of Five
qui intègre, en plus, des opérations d'échanges
d'emplacements mémoire (move semantic)
Elles sont liées aux références && dont nous avons un
peu parlé au cours 2, et qui sont hors programme
pour nous.