# CORRECTION D'EXAMEN 2022 - 2023

Grands Moulins

3/01/2023 - 9h30-11h30

Langage Objet Avancé - C++

**Consignes :**

- Terminez un nombre éventuellement limité d'exercice mais faites les soigneusement pour ne pas être pénalisé pour de l'inattention : il vaut mieux en faire un peu moins mais complètement, plutôt que d'essayer de tout faire maladroitement.
- Le barème est indicatif. **Aucun document n'est autorisé**.
- Les 4 premiers exercices sont un peu du type QCM (il faut quand même prendre le temps de la réflexion), les 2 derniers exercices sont plus ouverts, il vous faudra écrire du code.

# CORRECTION D'EXAMEN

Vous avez du vous rendre compte que certains "détails" sont importants en c++
Relisez vous, commentez vos choix, justifiez vos "hacks" invisibles.

Grands Moulins

3/01/2023 - 9h30-11h30

Objet Avancé - C++

**Consignes :**

- Terminez un nombre éventuellement limité d'exercice mais faites les soigneusement pour ne pas être pénalisé pour de l'inattention : il vaut mieux en faire un peu moins mais complètement, plutôt que d'essayer de tout faire maladroitement.
- Le barème est indicatif. **Aucun document n'est autorisé**.
- Les 4 premiers exercices sont un peu du type QCM (il faut quand même prendre le temps de la réflexion), les 2 derniers exercices sont plus ouverts, il vous faudra écrire du code.

# CORRECTION D'EXAMEN

Votre objectif est de garantir ~14 points.
Ne pensez à obtenir plus qu'après avoir sécurisé l'essentiel.

Grands Moulins

3/01/2023 - 9h30-11h30

ge Objet Avancé - C++

**Consignes :**

- Terminez un nombre éventuellement limité d'exercice mais faites les soigneusement pour ne pas être pénalisé pour de l'inattention : il vaut mieux en faire un peu moins mais complètement, plutôt que d'essayer de tout faire maladroitement.
- Le barème est indicatif. **Aucun document n'est autorisé**.
- Les 4 premiers exercices sont un peu du type QCM (il faut quand même prendre le temps de la réflexion), les 2 derniers exercices sont plus ouverts, il vous faudra écrire du code.

# CORRECTION D'EXAMEN

Un exercice baclé =
des petites choses correctes, ok, mais aussi
des énormités, une présentation illisible.
Le correcteur ne peut pas faire semblant de
les ignorer

Grands Moulins

Avancé - C++

**Consignes :**

- Terminez un nombre éventuellement limité d'exercice mais faites les soigneusement pour ne pas être pénalisé pour de l'inattention : il vaut mieux en faire un peu moins mais complètement, plutôt que d'essayer de tout faire maladroitement.
- Le barème est indicatif. **Aucun document n'est autorisé**.
- Les 4 premiers exercices sont un peu du type QCM (il faut quand même prendre le temps de la réflexion), les 2 derniers exercices sont plus ouverts, il vous faudra écrire du code.

# CORRECTION D'EXAMEN

Cela vous permet de vous organiser.

Il pourra être ajusté.
ex : si les meilleures notes sont autour de 16, le barème sera probablement remonté.

Grands Moulins

Avancé - C++

---

**Consignes :**

- Terminez un nombre éventuellement limité d'exercice mais faites les soigneusement pour ne pas être pénalisé pour d[illegible]attention : il vaut mieux en faire un peu moins mais complètement, plutôt que d'essayer de tout faire maladroitement.
- Le barème est indicatif. **Aucun document n'est autorisé**.
- Les 4 premiers exercices sont un peu du type QCM (il faut quand même prendre le temps de la réflexion), les 2 derniers exercices sont plus ouverts, il vous faudra écrire du code.

# CORRECTION D'EXAMEN

Il faut avoir en tête l'essentiel.
C'est un contrôle de connaissances...

Grands Moulins

3/01/2023 - 9h30-11h30

Lan[illegible] Avancé - C++

**Consignes :**

- Terminez un nombre éventuellement l[illegible] d'exercice mais faites les soigneusement pour ne pas être pénalisé pour de l'inattention : il vaut [illegible]ux en faire un peu moins mais complètement, plutôt que d'essayer de tout faire maladroitement.
- Le barème est indicatif. **Aucun document n'est autorisé**.
- Les 4 premiers exercices sont un peu du type QCM (il faut quand même prendre le temps de la réflexion), les 2 derniers exercices sont plus ouverts, il vous faudra écrire du code.

**Exercice 1 [3 points]**

Le code suivant comporte 6 erreurs détectées à la compilation (il n'y a pas d'erreur de syntaxe). Pouvez-vous les identifier par leur numéro de ligne ET indiquer par une **phrase claire** en quoi c'est une erreur.

```
1  class X {
   private :
3   const int value;
   public :
5   X(int v) : value {v} { value += 10; }
    const int &getValue () { return value; }
7  friend void f ();
   };
9
   void f () {
11 const X x {56};
   const int *pi;
13 int i = x.value;
   pi = &(x.getValue() ) ;
15 pi = &i ;
   *pi = 34;
17 const int &k = x.getValue ();
   };
19
   int main () {
21 X x (45);
   int i;
23 i = x.getValue();
   int &j = x.getValue();
25 i = x.value ;
   f();
27 return 0 ;
   };
```

**Exercice 1 [3 points]**

Le code suivant comporte 6 erreurs détectées à la compilation (il n'y a pas d'erreur de syntaxe). Pouvez-vous les identifier par leur numéro de ligne ET indiquer par une **phrase claire** en quoi c'est une erreur.

```
1  class X {
   private :
3   const int value;
   public :
5   X(int v) : value {v} { value += 10; }
    const int &getValue () { return value;
7  friend void f ();
   };
9
   void f () {
11 const X x {56};
   const int *pi;
13 int i = x.value;
   pi = &(x.getValue() ) ;
15 pi = &i ;
   *pi = 34;
17 const int &k = x.getValue ();
   };
19
   int main () {
21 X x (45);
   int i;
23 i = x.getValue();
   int &j = x.getValue();
25 i = x.value ;
   f();
27 return 0 ;
   };
```

ligne 5 :
value, déclarée const ne peut être modifiée

**Exercice 1 [3 points]**

Le code suivant comporte 6 erreurs détectées à la compilation (il n'y a pas d'erreur de syntaxe). Pouvez-vous les identifier par leur numéro de ligne ET indiquer par une **phrase claire** en quoi c'est une erreur.

```
1  class X {
   private :
3   const int value;
   public :
5   X(int v) : value {v} { value += 10; }
    const int &getValue () { return value; }
7  friend void f ();
   };
9
   void f () {
11 const X x {56};
   const int *pi;
13 int i = x.value;
   pi = &(x.getValue() ) ;
15 pi = &i ;
   *pi = 34;
17 const int &k = x.getValue ();
   };
19
   int main () {
21 X x (45);
   int i;
23 i = x.getValue();
   int &j = x.getValue();
25 i = x.value ;
   f();
27 return 0 ;
   };
```

ligne 14 (un peu difficile) : le pb est que x est const, mais que getValue() n'a pas const dans sa signature

**Exercice 1 [3 points]**

Le code suivant comporte 6 erreurs détectées à la compilation (il n'y a pas d'erreur de syntaxe). Pouvez-vous les identifier par leur numéro de ligne ET indiquer par une **phrase claire** en quoi c'est une erreur.

```
1  class X {
   private :
3   const int value;
   public :
5   X(int v) : value {v} { va
    const int &getValue () { retu
7  friend void f ();
};
9
void f () {
11 const X x {56};
   const int *pi;
13 int i = x.value;
   pi = &(x.getValue() ) ;
15 pi = &i ;
   *pi = 34;
17 const int &k = x.getValue ();
};
19
int main () {
21 X x (45);
   int i;
23 i = x.getValue();
   int &j = x.getValue();
25 i = x.value ;
   f();
27 return 0 ;
};
```

il aurait fallu :
const int & getValue() **const**

ligne 14 (un peu difficile) :
le pb est que x est const,
mais que getValue() n'a pas
const dans sa signature

**Exercice 1 [3 points]**

Le code suivant comporte 6 erreurs détectées à la compilation (il n'y a pas d'erreur de syntaxe). Pouvez-vous les identifier par leur numéro de ligne ET indiquer par une **phrase claire** en quoi c'est une erreur.

```
1  class X {
   private :
3   const int value;
   public :
5   X(int v) : value {v} { value += 10; }
    const int &getValue () { return value; }
7  friend void f ();
   };
9
   void f () {
11 const X x {56};
   const int *pi;
13 int i = x.value;
   pi = &(x.getValue() ) ;
15 pi = &i ;
   *pi = 34;
17 const int &k = x.getValue ();
   };
19
   int main () {
21 X x (45);
   int i;
23 i = x.getValue();
   int &j = x.getValue();
25 i = x.value ;
   f();
27 return 0 ;
   };
```

ligne 16 :
l'entier vu par pi est considéré constant. On ne peut pas le changer via pi.

**Exercice 1 [3 points]**

Le code suivant comporte 6 erreurs détectées à la compilation (il n'y a pas d'erreur de syntaxe). Pouvez-vous les identifier par leur numéro de ligne ET indiquer par une **phrase claire** en quoi c'est une erreur.

```
1  class X {
   private :
3   const int value;
   public :
5   X(int v) : value {v} { valu
    const int &getValue () { re
7  friend void f ();
};
9
void f () {
11 const X x {56};
   const int *pi;
13 int i = x.value;
   pi = &(x.getValue() ) ;
15 pi = &i ;
   *pi = 34;
17 const int &k = x.getValue ();
};
19
int main () {
21 X x (45);
   int i;
23 i = x.getValue();
   int &j = x.getValue();
25 i = x.value ;
   f();
27 return 0 ;
};
```

ligne 17 :
même erreur qu'en ligne 14.
On la remet car vous en cherchez un certain nb (6) et on veut des réponses bien assumées

**Exercice 1 [3 points]**

Le code suivant comporte 6 erreurs détectées à la compilation (il n'y a pas d'erreur de syntaxe). Pouvez-vous les identifier par leur numéro de ligne ET indiquer par une **phrase claire** en quoi c'est une erreur.

```
1  class X {
   private :
3   const int value;
   public :
5   X(int v) : value {v} { valu
    const int &getValue () { re
7  friend void f ();
};
9
void f () {
11 const X x {56};
   const int *pi;
13 int i = x.value;
   pi = &(x.getValue() ) ;
15 pi = &i ;
   *pi = 34;
17 const int &k = x.getValue ();
};
19
int main () {
21 X x (45);
   int i;
23 i = x.getValue();
   int &j = x.getValue();
25 i = x.value ;
   f();
27 return 0 ;
};
```

ligne 24 :
cette fois x est non const,
getValue retourne qq chose.
Mais c'est constant, et
l'alias prétend ne pas l'être

**Exercice 1 [3 points]**

Le code suivant comporte 6 erreurs détectées à la compilation (il n'y a pas d'erreur de syntaxe). Pouvez-vous les identifier par leur numéro de ligne ET indiquer par une **phrase claire** en quoi c'est une erreur.

```
1  class X {
   private :
3   const int value;
   public :
5   X(int v) : value {v} { valu
    const int &getValue () { re
7  friend void f ();
};
9
void f () {
11 const X x {56};
   const int *pi;
13 int i = x.value;
   pi = &(x.getValue() ) ;
15 pi = &i ;
   *pi = 34;
17 const int &k = x.getValue (
};
19
int main () {
21 X x (45);
   int i;
23 i = x.getValue();
   int &j = x.getV ue();
25 i = x.value ;
   f();
27 return 0 ;
};
```

ligne 25 :
l'attribut value est private.
Accès interdit ici.

**Exercice 2 [2 points]**

Le code suivant compile, mais produit à l'exécution un segmentation fault. Pouvez vous expliquer pourquoi ?

```
class A {};
class B {
 private :
  A* p;
 public :
  B();
  ~B();
};

B::B():p{new A()} {}
B::~B() {delete p;}

void f(B b) {}

int main ( ) {
 B b;
 f(b);
};
```

**Exercice 2 [2 points]**

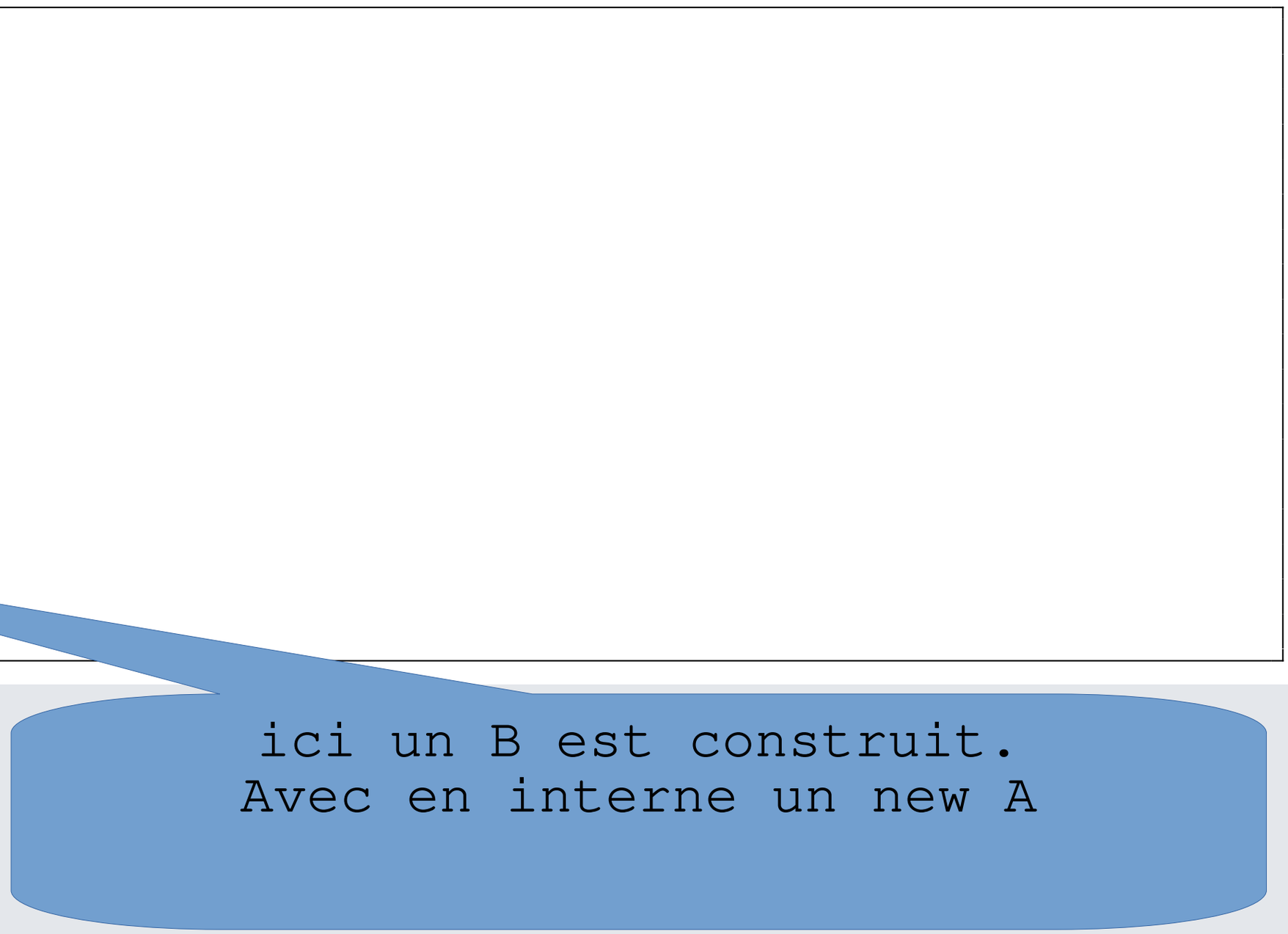
Le code suivant compile, mais produit à l'exécution un segmentation fault. Pouvez vous expliquer pourquoi ?

```
class A {};
class B {
 private :
  A* p;
 public :
  B();
  ~B();
};

B::B():p{new A()} {}
B::~B() {delete p;}

void f(B b) {}

int main ( ) {
 B b;
 f(b);
};
```



**Exercice 2 [2 points]**

Le code suivant compile, mais produit à l'exécution un segmentation fault. Pouvez vous expliquer pourquoi ?

```
class A {};
class B {
 private :
  A* p;
 public :
  B();
  ~B();
};

B::B():p{new A()} {}
B::~B() {delete p;}

void f(B b) {}

int main ( ) {
 B b;
 f(b);
};
```

Cette signature, signifie qu'une copie du paramètre est faite.

**Exercice 2 [2 points]**

Le code suivant compile, mais produit à l'exécution un segmentation fault. Pouvez vous expliquer pourquoi ?

```
class A {};
class B {
 private :
  A* p;
 public :
  B();
  ~B();
};

B::B():p{new A()} {}
B::~B() {delete p;}

void f(B b) {}

int main ( ) {
 B b;
 f(b);
};
```

Le b local est obtenu grâce au constructeur de copie par défaut. Il y reporte l'attribut p tel quel.

**Exercice 2 [2 points]**

Le code suivant compile, mais produit à l'exécution un segmentation fault. Pouvez vous expliquer pourquoi ?

```
class A {};
class B {
 private :
  A* p;
 public :
  B();
  ~B();
};

B::B():p{new A()} {}
B::~B() {delete p;}

void f(B b) {}

int main ( ) {
 B b;
 f(b);
};
```

à la sortie du bloc, le b local est détruit.

**Exercice 2 [2 points]**

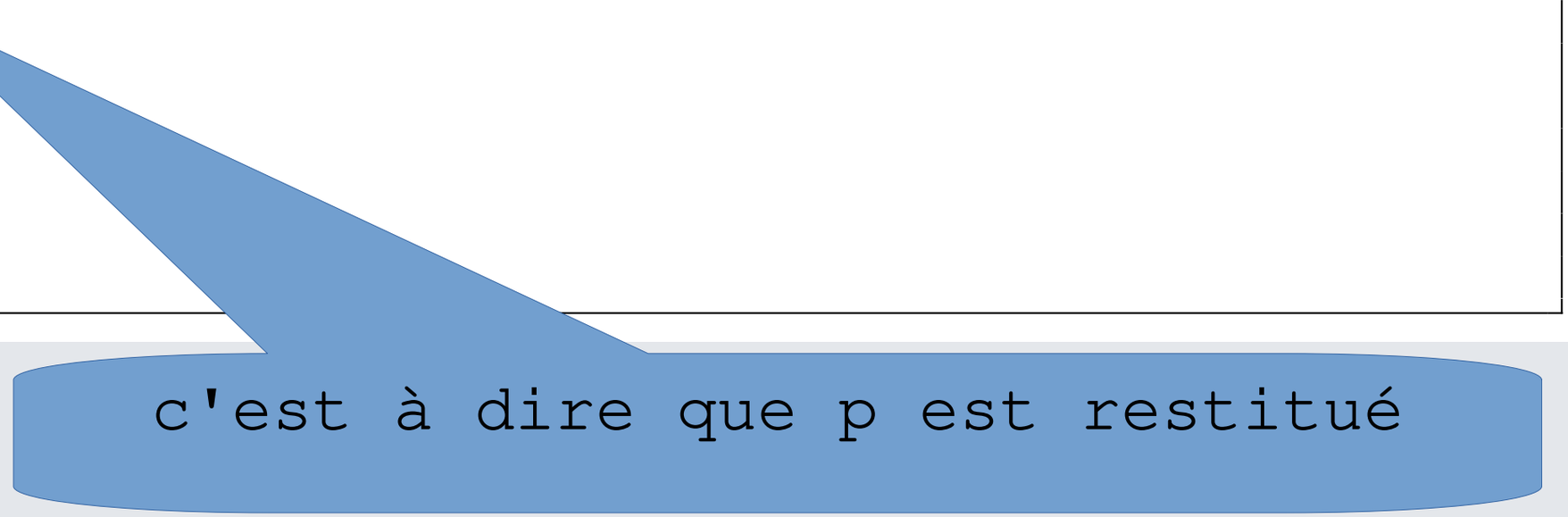
Le code suivant compile, mais produit à l'exécution un segmentation fault. Pouvez vous expliquer pourquoi ?

```
class A {};
class B {
 private :
  A* p;
 public :
  B();
  ~B();
};

B::B():p{new A()} {}
B::~B() {delete p;}

void f(B b) {}

int main ( ) {
 B b;
 f(b);
};
```



**Exercice 2 [2 points]**

Le code suivant compile, mais produit à l'exécution un segmentation fault. Pouvez vous expliquer pourquoi ?

```
class A {};
class B {
 private :
  A* p;
 public :
  B();
  ~B();
};

B::B():p{new A()} {}
B::~B() {delete p;}

void f(B b) {}

int main ( ) {
 B b;
 f(b);
};
```

f est terminé sans erreur

**Exercice 2 [2 points]**

Le code suivant compile, mais produit à l'exécution un segmentation fault. Pouvez vous expliquer pourquoi ?

```
class A {};
class B {
 private :
  A* p;
 public :
  B();
  ~B();
};

B::B():p{new A()} {}
B::~B() {delete p;}

void f(B b) {}

int main ( ) {
 B b;
 f(b);
};
```

à la sortie du main, b est détruit : son p (déjà deleted) est à nouveau deleted. C'est ça qui produit le 'core dumped'

**Exercice 3 [3 points]**

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { return g() + " by f() in D"; }
  string g() const { return "g() in D"; }
  string h() const { return g() + " by h() in D"; }
};

class E : public D {
 public :
  string f() const { return g() + " by f() in E";}
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;  // (1)
 cout << y.h() << endl;  // (2)
 D* z= new E{};
 cout << z->f() << endl; // (3)
 cout << z->h() << endl; // (4)
 B x = D{};
 cout << x.f() << endl;  // (5)
 cout << x.h() << endl;  // (6)
}
```

**Exercice 3 [3 points]**

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { return g() + " by f() in D"; }
  string g() const { return "g() in D"; }
  string h() const { return g() + " by h() in D"; }
};

class E : public D {
 public :
  string f() const { return
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;  // (1)
 cout << y.h() << endl;  // (2)
 D* z= new E{};
 cout << z->f() << endl; // (3)
 cout << z->h() << endl; // (4)
 B x = D{};
 cout << x.f() << endl;  // (5)
 cout << x.h() << endl;  // (6)
}
```

y est typé B (mais est un D)
f n'est pas virtual : celui de B

**Exercice 3 [3 points]**

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { retu          " by f() in D"; }
  string g() const { return "g()
  string h() const { return g() +          D"; }
};

class E : public D {
 public :
  string f() const { return
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;
 cout << y.h() << endl;   // (
 D* z= new E{};
 cout << z->f() << endl; // (3)
 cout << z->h() << endl; // (4)
 B x = D{};
 cout << x.f() << endl;  // (5)
 cout << x.h() << endl;  // (6)
}
```

y est typé B (mais est un D)
f n'est pas virtual : celui de B
g est virtual dans B donc passe à D

**Exercice 3 [3 points]**

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { return g() + " by f() in D"; }
  string g() const { return "g() in D"; }
  string h() const { return g() + " by h() in D"; }
};

class E : public D {
 public :
  string f() const { return g()
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;
 cout << y.h() << endl;
 D* z= new E{};
 cout << z->f() << endl; //
 cout << z->h() << endl; // (4)
 B x = D{};
 cout << x.f() << endl;  // (5)
 cout << x.h() << endl;  // (6)
}
```

y est typé B (mais est un D)
f n'est pas virtual : celui de B
g est virtual dans B donc passe à D
(1) g() in D by f() in B

**Exercice 3 [3 points]**

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { return g() + " by f() in D"; }
  string g() const { return "g() in D"; }
  string h() const { return g() + " by h() in D"; }
};

class E : public D {
 public :
  string f() const { return
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;  // (1)
 cout << y.h() << endl;  // (2)
 D* z= new E{};
 cout << z->f() << endl; // (3)
 cout << z->h() << endl; // (4)
 B x = D{};
 cout << x.f() << endl;  // (5)
 cout << x.h() << endl;  // (6)
}
```

y est typé B (mais est un D)
h est virtual : celui de D

**Exercice 3 [3 points]**

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { return g() + " by f() in D"; }
  string g() const { return "g() in D"; }
  string h() const { return g() + " by h() in D"; }
};

class E : public D {
 public :
  string f() const { return
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;
 cout << y.h() << endl;   // (
 D* z= new E{};
 cout << z->f() << endl; // (3)
 cout << z->h() << endl; // (4)
 B x = D{};
 cout << x.f() << endl;  // (5)
 cout << x.h() << endl;  // (6)
}
```

y est typé B (mais est un D)
h est virtual : celui de D
g (sur this) est virtual : D

**Exercice 3 [3 points]**

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { return g() + " by f() in D"; }
  string g() const { return "g() in D"; }
  string h() const { return g() + " by h() in D"; }
};

class E : public D {
 public :
  string f() const { return
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;
 cout << y.h() << endl;
 D* z= new E{};
 cout << z->f() << endl; // (3)
 cout << z->h() << endl; // (4)
 B x = D{};
 cout << x.f() << endl;  // (5)
 cout << x.h() << endl;  // (6)
}
```

y est typé B (mais est un D)
h est virtual : celui de D
g (sur this) est virtual : D
(2) g() in D by h() in D

**Exercice 3 [3 points]**

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { return g() + " by f() in D"; }
  string g() const { return "g() in D"; }
  string h() const { return g() + " by h() in D"; }
};

class E : public D {
 public :
  string f() const { return g() + " by f() in E";}
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;
 cout << y.h() << endl;
 D* z= new E{};
 cout << z->f() << endl; // (3)
 cout << z->h() << endl; // (4)
 B x = D{};
 cout << x.f() << endl;  // (5)
 cout << x.h() << endl;  // (6)
}
```

z est typé D (mais est un E)
f est virtual dans D : celui de E

**Exercice 3 [3 points]**

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { return g() + " by f() in D"; }
  string g() const { return "g() in D"; }
  string h() const { return g() + " by h() in D"; }
};

class E : public D {
 public :
  string f() const { return g() + " by f() in E";}
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;
 cout << y.h() << endl;
 D* z= new E{};
 cout << z->f() << endl;
 cout << z->h() << endl; // (
 B x = D{};
 cout << x.f() << endl;  // (5)
 cout << x.h() << endl;  // (6)
}
```

z est typé D (mais est un E)
f est virtual après D : celui de E
g est dispo dans D

**Exercice 3 [3 points]**

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { return g() + " by f() in D"; }
  string g() const { return "g() in D"; }
  string h() const { return g() + " by h() in D"; }
};

class E : public D {
 public :
  string f() const { return g() + " by f() in E";}
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;
 cout << y.h() << endl;
 D* z= new E{};
 cout << z->f() << endl;
 cout << z->h() << endl;
 B x = D{};
 cout << x.f() << endl;   // (5)
 cout << x.h() << endl;   // (6)
}
```

z est typé D (mais est un E)
f est virtual après D : celui de E
g est dispo dans D
(3) g() in D by f() in E

**Exercice 3 [3 points]**

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { return g() + " by f() in D"; }
  string g() const { return "g() in D"; }
  string h() const { return g() + " by h() in D"; }
};

class E : public D {
 public :
  string f() const { return g() + " by f() in E";}
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;
 cout << y.h() << endl;
 D* z= new E{};
 cout << z->f() << endl; // (3)
 cout << z->h() << endl; // (4)
 B x = D{};
 cout << x.f() << endl;  // (5)
 cout << x.h() << endl;  // (6)
}
```

z est typé D (mais est un E)
h est dispo dans D

**Exercice 3 [3 points]**

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { return g() + " by f() in D"; }
  string g() const { return "g() in D"; }
  string h() const { return g() + " by h() in D"; }
};

class E : public D {
 public :
  string f() const { return g() +          n E";}
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;
 cout << y.h() << endl;
 D* z= new E{};
 cout << z->f() << endl;
 cout << z->h() << endl; //
 B x = D{};
 cout << x.f() << endl;  // (5)
 cout << x.h() << endl;  // (6)
}
```

z est typé D (mais est un E)
h est dispo dans D
g est dispo dans D

**Exercice 3 [3 points]**

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { return g() + " by f() in D"; }
  string g() const { return "g() in D"; }
  string h() const { return g() + " by h() in D"; }
};

class E : public D {
 public :
  string f() const { return g() +          n E";}
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;
 cout << y.h() << endl;
 D* z= new E{};
 cout << z->f() << endl;
 cout << z->h() << endl;
 B x = D{};
 cout << x.f() << endl;   // (5)
 cout << x.h() << endl;   // (6)
}
```

z est typé D (mais est un E)
h est dispo dans D
g est dispo dans D
(4) g() in D by h() in D

**Exercice 3 [3 points]**

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { return g() + " by f() in D"; }
  string g() const { return "g() in D"; }
  string h() const { return g() + " by h() in D"; }
};

class E : public D {
 public :
  string f() const { retu
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;
 cout << y.h() << endl;
 D* z= new E{};
 cout << z->f() << endl; // (3)
 cout << z->h() << endl; // (4)
 B x = D{};
 cout << x.f() << endl;  // (5)
 cout << x.h() << endl;  // (6)
}
```

x est typé B, et est bien un B !
car un D anonyme est construit,
puis le constructeur par copie
de B est invoqué pour x

**Exercice 3 [3 points]**

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { return g() + " by f() in D"; }
  string g() const { return "g() in D"; }
  string h() const { return g() + " by h() in D"; }
};

class E : public D {
 public :
  string f() const { retu
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;   // (1)
 cout << y.h() << endl;   //
 D* z= new E{};
 cout << z->f() << en     // (3)
 cout << z->h() <<  ndl; // (4)
 B x = D{};
 cout << x.f() << endl;   // (5)
 cout << x.h() << endl;   // (6)
}
```

x est typé B, et est bien un B !
f de B

**Exercice 3 [3 points]**

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { retu          + " by f() in D"; }
  string g() const { return "g()
  string h() const { return g() +          D"; }
};

class E : public D {
 public :
  string f() const { retu
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;   //
 cout << y.h() << endl;   // (2)
 D* z= new E{};
 cout << z->f() << endl; // (3)
 cout << z->h() << endl; // (4)
 B x = D{};
 cout << x.f() << endl;   // (5)
 cout << x.h() << endl;   // (6)
}
```

x est typé B, et est bien un B !
f de B
puis g de B

**Exercice 3 [3 points]**

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { retu          + " by f() in D"; }
  string g() const { return "g()
  string h() const { return g() +          D"; }
};

class E : public D {
 public :
  string f() const { retur
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;
 cout << y.h() << endl;
 D* z= new E{};
 cout << z->f() << endl; // (3)
 cout << z->h() << endl; // (4)
 B x = D{};
 cout << x.f() << endl;  // (5)
 cout << x.h() << endl;  // (6)
}
```

x est typé B, et est bien un B !
f de B
puis g de B
(5) g() in B by f() in B

**Exercice 3 [3 points]**

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { return g() + " by f() in D"; }
  string g() const { return "g() in D"; }
  string h() const { return g() + " by h() in D"; }
};

class E : public D {
 public :
  string f() const { retu
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;   // (1)
 cout << y.h() << endl;   // (2)
 D* z= new E{};
 cout << z->f() << endl;
 cout << z->h() << endl;  (4)
 B x = D{};
 cout << x.f() << endl;   // (5)
 cout << x.h() << endl;   // (6)
}
```

x est typé B, et est bien un B !
h de B

**Exercice 3 [3 points]**

Le programme suivant compile et s'exécute sans problèmes. Quels sont les 6 affichages produits par le main ? (Il y a peut être un ou deux endroits qui méritent des explications)

```
class B {
 public:
  string f() const { return g() + " by f() in B"; }
  virtual string g() const { return "g() in B"; }
  virtual string h() const { return g() + " by h() in B"; }
};

class D : public B {
 public:
  virtual string f() const { return g
  string g() const { return "g() in D
  string h() const { return g() + "
};

class E : public D {
 public :
  string f() const { retur
};

int main() {
 B & y= *new D{};
 cout << y.f() << endl;
 cout << y.h() << endl;
 D* z= new E{};
 cout << z->f() << endl; // (3)
 cout << z->h() << endl; // (4)
 B x = D{};
 cout << x.f() << endl;  // (5)
 cout << x.h() << endl;  // (6)
}
```

by f() in D"; }

D"; }

x est typé B, et est bien un B !
h de B
puis g de B
(6) g() in B by h() in B

**Exercice 4 [3 points]**

Voici deux situations qu'on veut absolument représenter à l'aide de l'**héritage multiple** :

- Au niveau des Animaux on stocke un attribut pour mémoriser un ADN. Parmi les Animaux on distingue la sous-classe des Carnivores, et celle des Herbivores. Les Omnivores sont eux à la fois des Carnivores et des Herbivores. Vous partirez de :

```
class Animal {
 public :
  const string adn;
  Animal (string x) :adn {x} {}
};
```

Ecrivez les classes Carnivore, Herbivore, Omnivore ainsi qu'un exemple dans un main de construction et d'affichage de l'adn unique d'un omnivore particulier.

- Au niveau des Composants on stocke un attribut pour mémoriser un numéro de série. Parmi les Composants on distingue les sous-classes Clavier et Ecran. Par héritage un Ordi est à la fois un Clavier et un Ecran. Vous partirez de :

```
class Composant {
 private :
  static int NB;
 public :
  const string num;
  Composant ():num{to_string(NB++)}{}
};
int Composant::NB=0;
```

qui permet une numérotation automatique à partir d'un compteur statique.

Ecrivez les classes Ecran, Clavier, Ordi ainsi qu'un exemple dans un main de construction d'un ordi et d'affichage des numéros de série différents de son clavier et de son écran.

**Exercice 4 [3 points]**

Voici deux situations qu'on veut absolument représenter à l'aide de l'**héritage multiple** :

- Au niveau des Animaux on stocke un attribut pour mémoriser un ADN. Parmi les Animaux on distingue la sous-classe des Carnivores, et celle des Herbivores. Les Omnivores sont eux à la fois des Carnivores et des Herbivores. Vous partirez de :

```
class Animal {
 public :
  const string adn;
  Animal (string x) :adn{x} {}
};
```

Ecrivez les classes Carn
d'affichage de l'adn un

un animal n'a qu'un seul adn.
L'omnivore, est un animal indirect
via carnivore ou via herbivore,
qui ne doit avoir qu'une seule
base animal !

- Au niveau des Compos
Composants on distingu
et un Ecran. Vous partire

```
class Composant {
 private :
  static int NB;
 public :
  const string num;
  Composant ():num{to_string(NB++)}{}
};
int Composant::NB=0;
```

qui permet une numérotation automatique à partir d'un compteur statique.

Ecrivez les classes Ecran, Clavier, Ordi ainsi qu'un exemple dans un main de construction d'un ordi et d'affichage des numéros de série différents de son clavier et de son écran.

```
class Animal {
public :
  const string adn;
  Animal (string x) :adn {x} {}
};

class Herbivore : virtual public Animal {
 public :
  Herbivore(string x) : Animal{x} {}
};

class Carnivore : virtual public Animal {
public :
    Carnivore(string x): Animal{x} {}
};
class Omnivore : public Herbivore, public Carnivore {
public :
    Omnivore(string x)
                :Animal{x}, Herbivore{x}, Carnivore{x} {}
};
```

```
int main() {
    Omnivore o{"poulet"};
    cout << o.adn << endl;
}
```

- Au niveau des Composants on stocke un attribut pour mémoriser un numéro de série. Parmi les Composants on distingue les sous-classes Clavier et Ecran. Par héritage un Ordi est à la fois un Clavier et un Ecran. Vous partirez de :

```
class Composant {
 private :
  static int NB;
 public :
  const string num;
  Composant ():num{to_string(NB++)}{}
};
int Composant::NB=0;
```

qui permet une numérotation automatique à partir d'un compteur statique.

Ecrivez les classes Ecran, Clavier, Ordi ainsi qu'un exemple dans un main de construction d'un ordi et d'affichage des numéros de série différents de son clavier et de son écran.

- Au niveau des Com
Composants on distin
et un Ecran. Vous par

Un ordi, qui hérite de clavier et d'écran, se compose de ces deux éléments. Il possède ces deux numéros de série. Il n'en a pas juste un qui le caractériserait.

```
class Composant {
 private :
  static int NB;
 public :
  const string num;
  Composant ():num{to_string(NB++)}{}
};
int Composant::NB=0;
```

qui permet une numérotation automatique à partir d'un compteur statique.

Ecrivez les classes Ecran, Clavier, Ordi ainsi qu'un exemple dans un main de construction d'un ordi et d'affichage des numéros de série différents de son clavier et de son écran.

```
class Composant {
private :
   static int NB;
public :
    const string num;
    Composant ():num{to_string(NB++)}{}
};
int Composant::NB=0;


class Clavier : public Composant {};
class Ecran : public Composant{};

class Ordi : public Clavier, public Ecran {};
```

```
int main() {
    Ordi x;
    cout << x.Clavier::num << " " << x.Ecran::num << endl;
}
```

On rappelle rapidement comment fonctionne un itérateur sur les collections en général en l'illustrant sur la classe vector. Chaque collection définit deux classes internes iterator et reverse_iterator. Dans l'exemple suivant, v.begin(),v.end() et v.rbegin(), v.rend() en sont respectivement des instances.

```
vector<int> v {1,2,3};  // pour l'exemple
// pour une iteration de gauche a droite :
for(vector<int>::iterator i{v.begin()} ; i != v.end(); ++i) cout << *i;
cout << endl;
// pour une iteration de droite a gauche :
for(vector<int>::reverse_iterator i{v.rbegin()} ; i != v.rend(); ++i) cout << *i;
cout << endl;
```

L'affichage résultant est

```
123
321
```

D'un autre coté nous avons aussi la possibilité d'écrire :

```
for (int i: v) cout << i;
```

la syntaxe du for avec les deux points est une sorte de macro syntaxique qui est traduite vers la première des formes que nous avons données ci dessus, ligne 2-3 : le type iterator implicitement utilisé est celui du type interne de la collection parcourue, begin() et end() seront les méthodes invoquées pour obtenir les bornes. Ce qui est affiché est ici : 123, dans le sens "gauche-droite".

On souhaite introduire une forme syntaxique maison pour parcourir simplement notre vecteur dans l'autre sens, mais il n'y a pas de moyen de redéfinir ce qu'il se passe avec le symbole ":" on penche donc pour une solution qui aurait cette forme :

```
for (int i:wrap{v}) { cout << i ;}
```

Proposez une solution en ne vous occupant que du cas des vecteurs d'entiers (c.à d ne cherchez pas à faire de templates).

Indications :

- wrap est une nouvelle classe, qu'il vous faut écrire
- pour que le code soit lisible sur votre copie, écrivez définition et déclaration ensemble (ne séparez pas hpp et cpp)
- soyez rassurés : la macro for ( : ) se comportera avec wrap de la même façon qu'elle le fait pour les vector ou pour une autre une collection.

On rappelle rapidement comment fonctionne un itérateur sur les collections en général en l'illustrant sur la classe vector. Chaque collection définit deux classes internes iterator et reverse_iterator. Dans l'exemple suivant, v.begin(),v.end() et v.rbegin(), v.rend() en sont respectivement des instances.

```
vector<int> v {1,2,3};  // pour l'exemple
// pour une iteration de gauche a droite :
for(vector<int>::iterator i{v.begin()} ; i != v.end(); ++i) cout << *i;
cout << endl;
// pour une iterati   de droite a gauche :
for(vector<int>::rev    iterator i{v.rbegin()} ; i != v.rend(); ++i) cout << *i;
cout << endl;
```

L'affichage résultant est

```
123
321
```

D'un autre coté nous av

```
for (int i: v) cout
```

la syntaxe du for avec
des formes que nous
du type interne de la
bornes. Ce qui est aff

On souhaite introd
l'autre sens, mais il n'y a pa
pour une solution qui aurait cette forme :

```
for (int i:wrap{v}) { cout << i ;}
```

L'énoncé fait tous les rappels sur itérator : sa nature (une classe interne), la façon d'en obtenir les bornes (begin, end), ses opérateurs (++, * , !=)

Proposez une solution en ne vous occupant que du cas des vecteurs d'entiers (c.à d ne cherchez pas à faire de templates).

Indications :

- wrap est une nouvelle classe, qu'il vous faut écrire
- pour que le code soit lisible sur votre copie, écrivez définition et déclaration ensemble (ne séparez pas hpp et cpp)
- soyez rassurés : la macro for ( : ) se comportera avec wrap de la même façon qu'elle le fait pour les vector ou pour une autre une collection.

On rappelle rapidement comment fonctionne un itérateur sur les collections en général en l'illustrant sur la classe vector. Chaque collection définit deux classes internes iterator et reverse_iterator. Dans l'exemple suivant, v.begin(),v.end() et v.rbegin(), v.rend() en sont respectivement des instances.

```
vector<int> v {1,2,3};  // pour l'exemple
// pour une iteration de gauche a droite :
for(vector<int>::iterator i{v.begin()} ; i != v.end(); ++i) cout << *i;
cout << endl;
// pour une iteration de droite a gauche :
for(vector<int>::reverse_iterator i{v.rbegin()} ; i != v.rend(); ++i) cout << *i;
cout << endl;
```

L'affichage résultant est

```
123
321
```

D'un autre coté nous a

```
for (int i: v) cout
```

la syntaxe du for avec
des formes que nous
du type interne de la c
bornes. Ce qui est affiché est ici : 123, dans le sens "gauche-droite".

On souhaite introduire une forme syntaxique maison pour parcourir simplement notre vecteur dans l'autre sens, mais il n'y a pas de moyen de redéfinir ce qu'il se passe avec le symbole ":" on penche donc pour une solution qui aurait cette forme :

```
for (int i:wrap{v}) { cout << i ;}
```

On précise encore qu'il existe un objet qui fait déjà ce qu'on veux, mais il ne porte pas le bon nom ...

Proposez une solution en ne vous occupant que du cas des vecteurs d'entiers (c.à d ne cherchez pas à faire de templates).

Indications :

- wrap est une nouvelle classe, qu'il vous faut écrire
- pour que le code soit lisible sur votre copie, écrivez définition et déclaration ensemble (ne séparez pas hpp et cpp)
- soyez rassurés : la macro for ( : ) se comportera avec wrap de la même façon qu'elle le fait pour les vector ou pour une autre une collection.

```
class wrap {
      vector<int> &obj; // une référence -> pas de copie
public :
    class iterator {  // la classe interne
        ...
    };

    wrap(vector<int> &obj) : obj{ obj } {}
    iterator begin() { return iterator{...}; }
    iterator end() { return iterator{...};  }
};
```

```
class wrap {
      vector<int> &obj;
public :
    class iterator {
        ...
    };

    wrap(vector<int> &obj) : obj{ obj } {}
    iterator begin() { return iterator{obj.rbegin()}; }
    iterator end() { return iterator{obj.rend()};  }
};
```

On va simplement "adapter" le reverse itérator présent dans vector.

```
class wrap {
      vector<int> &obj;
public :
    class iterator {
        ...
    };

    wrap(vector<int> &obj) : obj{ obj } {}
    iterator begin() { return iterator{obj.rbegin()}; }
    iterator end() { return iterator{obj.rend()};  }
};
```

On va simplement "adapter" le reverse itérator présent dans vector.
Remarquez qu'on le construit à partir d'un objet anonyme.

```
class wrap {
      vector<int> &obj;
public :
    class iterator {
        vector<int>::reverse_iterator r;  // et pas &r
    public :
        iterator(vector<int>::reverse_iterator x):r{x} {}
        // reste à définir les opérations
    };

   public:
      wrap(vector<int> &obj) : obj{ obj } {}
      iterator begin() { return iterator{obj.rbegin()}; }
      iterator end() { return iterator{obj.rend()};   }
};
```

```
class wrap {
      vector<int> &obj;
public :
    class iterator {
        vector<int>::reverse_iterator r;  // et pas &r
    public :
        iterator(vector<int>::reverse_iterator x):r{x} {}
        int operator*(){ return *r;}
        ...
    };

   public:
      wrap(vector<int> &obj) : obj{ obj } {}
      iterator begin() { return iterator{obj.rbegin()}; }
      iterator end() { return iterator{obj.rend()};  }
};
```

```
class wrap {
      vector<int> &obj;
public :
    class iterator {
        vector<int>::reverse_iterator r;  // et pas &r
    public :
        iterator(vector<int>::reverse_iterator x):r{x} {}
        int operator*(){ return *r;}
        iterator&  operator++() { ++r; return *this;}
        ...
    };

  public:
      wrap(vector<int> &obj) : obj{ obj } {}
      iterator begin() { return iterator{obj.rbegin()}; }
      iterator end() { return iterator{obj.rend()};  }
};
```

```
class wrap {
      vector<int> &obj;
public :
    class iterator {
        vector<int>::reverse_iterator r;  // et pas &r
    public :
        iterator(vector<int>::reverse_iterator x):r{x} {}
        int operator*(){ return *r;}
        iterator&  operator++() { r++; return *this;}
        bool operator==(const iterator & iter) const
{ return r == iter.r; }
        bool operator!=(const iterator& iter) const
{  return r != iter.r; }
    };

   public:
      wrap(vector<int> &obj) : obj{ obj } {}
      iterator begin() { return iterator{obj.rbegin()}; }
      iterator end() { return iterator{obj.rend()};  }
};
```

```
int main ( ) {
    vector<int> v {1,2,3};

    wrap x {v};
    for (int i:x) { cout << i ;}

    // revient à

    for( wrap::iterator i{x.begin()} ; i != x.end(); ++i)
            cout << *i;

};
```

**Exercice 6 [5 points]**

Afin de mettre en pratique le patron Décorateur, nous allons concevoir une application qui permet de gérer la vente de desserts. Elle doit permettre d'afficher le nom complet du dessert (avec ses options) et d'obtenir son prix total. Les clients (qu'on ne représentera pas) ont potentiellement le choix entre plusieurs desserts de base (disons au moins crêpes ou gaufres). Sur chaque base ils peuvent ajouter un nombre quelconque d'ingrédients, par exemple du chocolat ou de la chantilly. Nous dirons qu'une crêpe (nature) coûte 1.50 euros et une gaufre (nature) 1.80 euros. L'ajout de chocolat sera facturé 0.20 euros et 0.50 euros pour de la chantilly.

1. Reprenez l'UML de figure 1 en l'adaptant aux desserts. Faites apparaître en particulier : les classes et attributs qui vous semblent indispensables (précisez leur visibilité), les constructeurs significatifs, les méthodes permettant d'obtenir nom et prix avec leurs spécifications exactes (virtuelle, abstraite, const ...).

2. Ecrivez le code correspondant (Ne séparez pas hpp et cpp pour que cela reste lisible. Pour info ma correction fait une trentaine de courtes lignes)

3. Ecrivez un main vous permettant d'afficher la description et le prix d'une crèpe au chocolat et à la chantilly avec un second supplément chocolat.

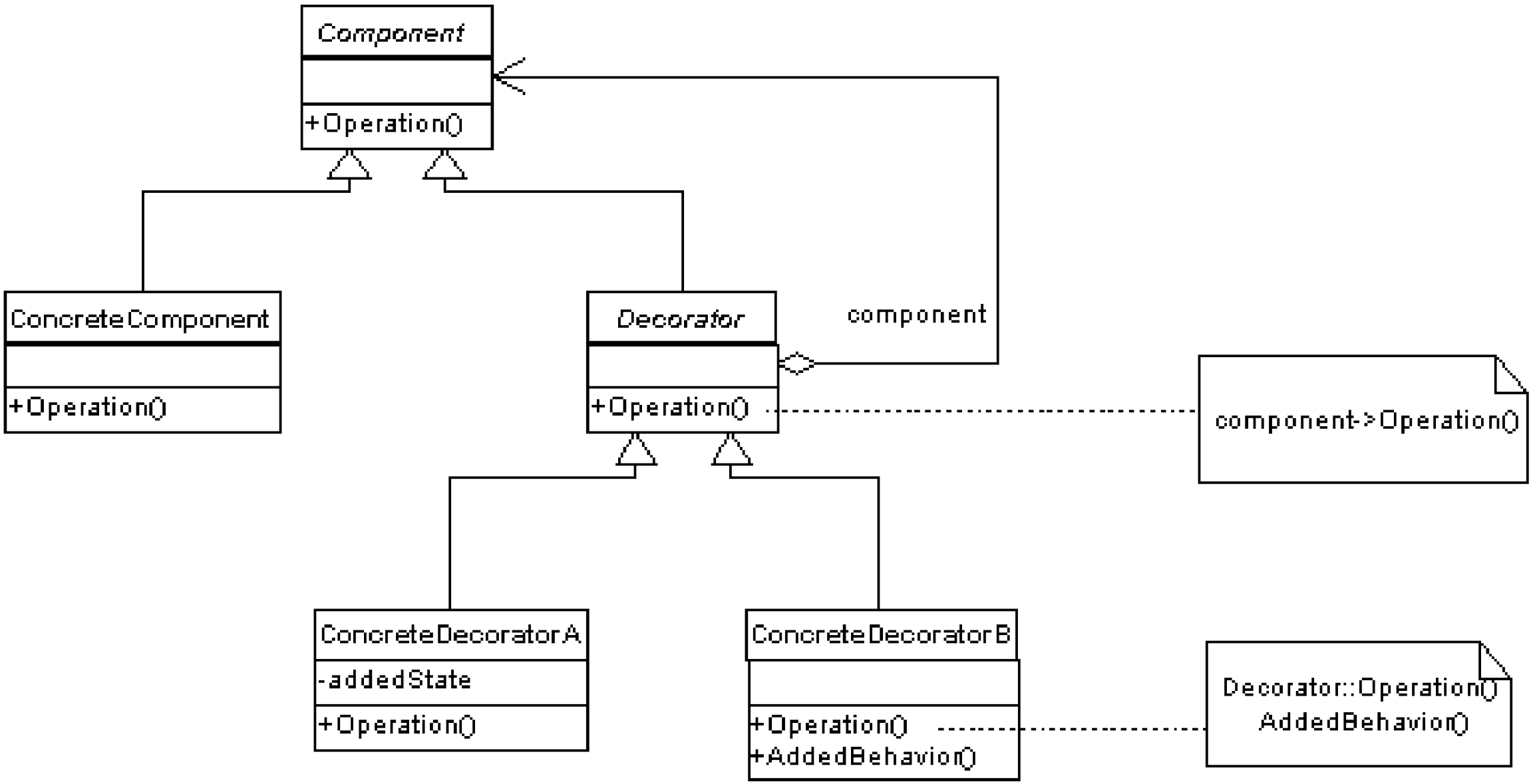



Figure 1: Rappel UML du patron décorateur.

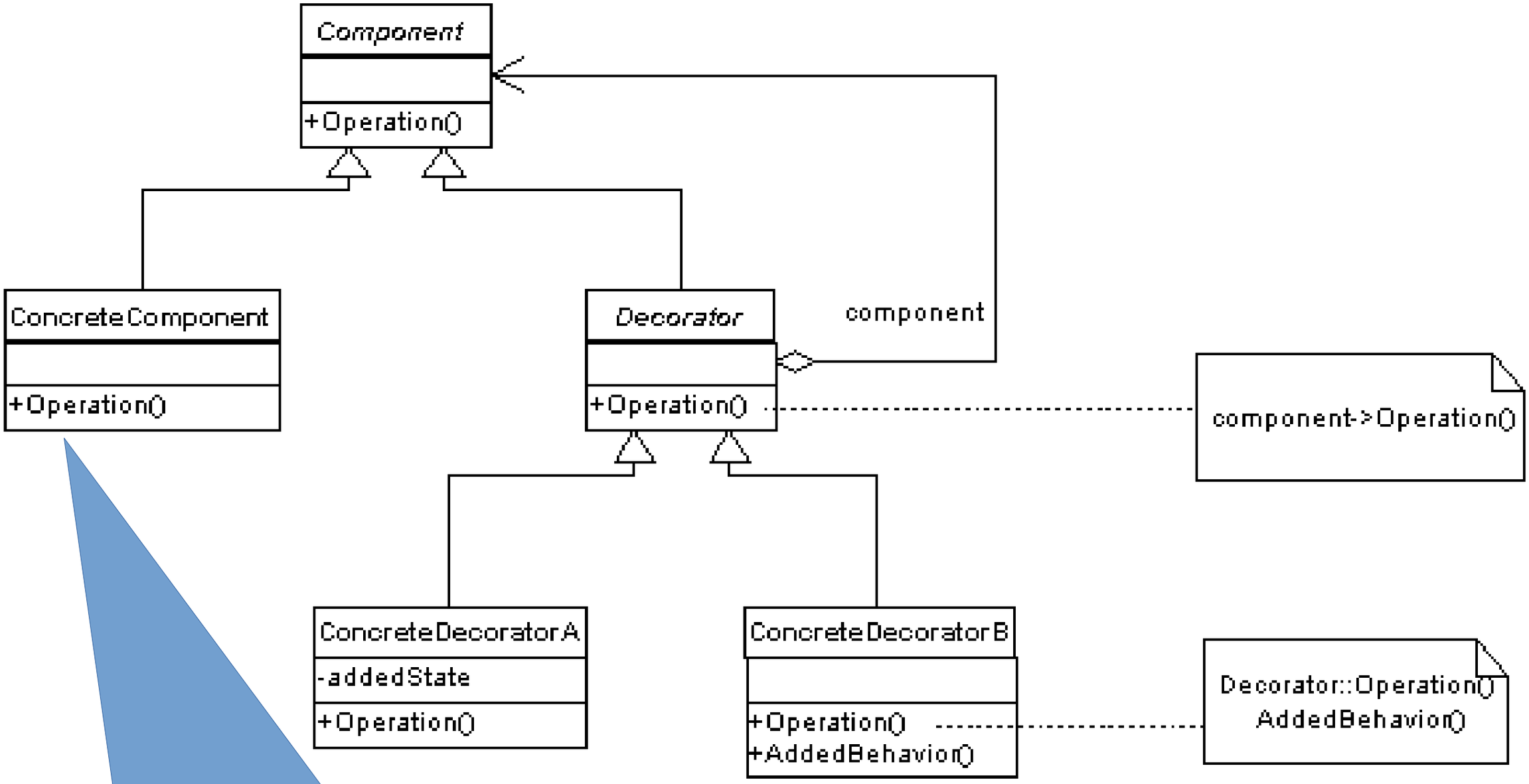



Figure 1: Rappel UML du patron décorateur.

2 composants concrets : crêpe ou gaufre

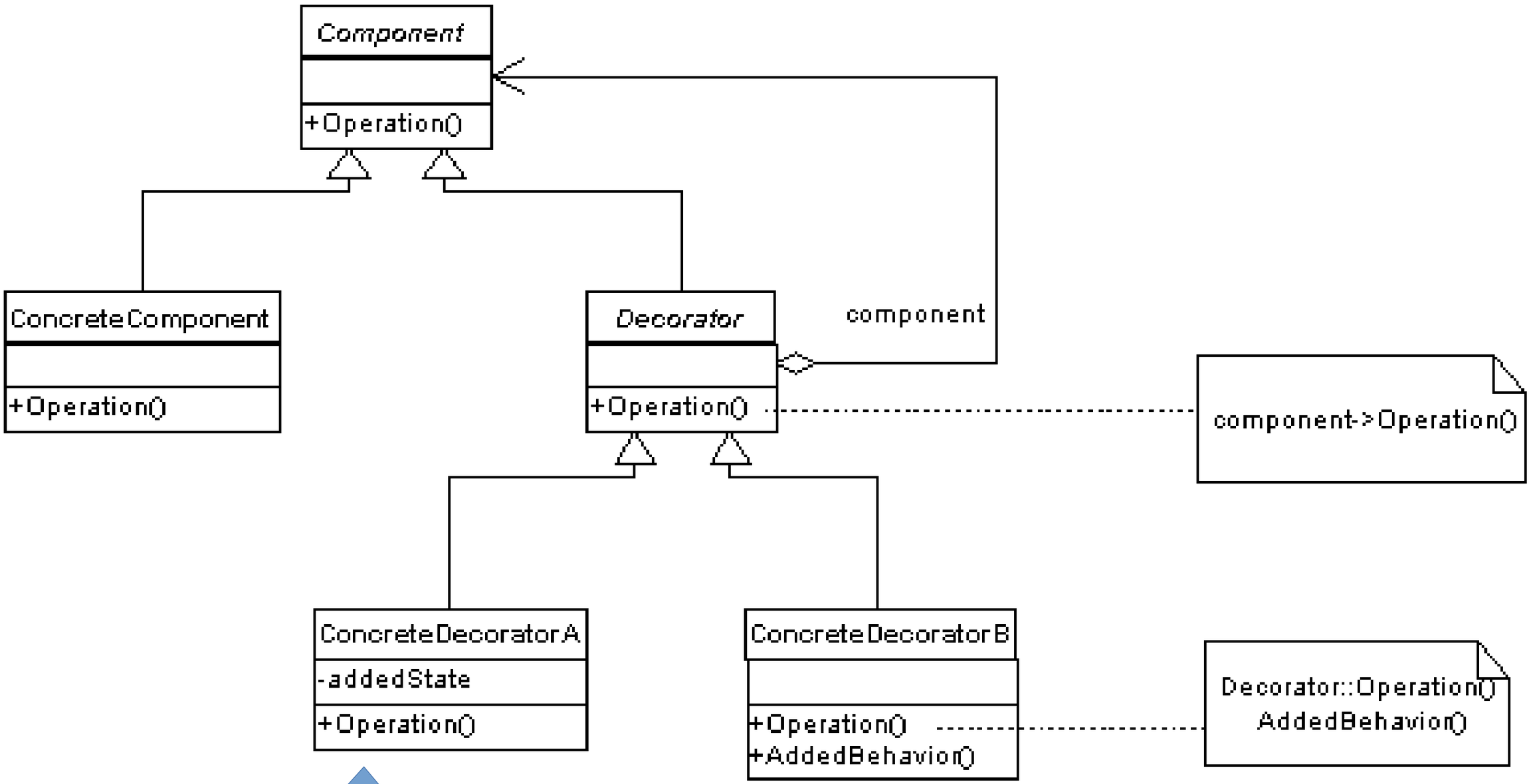



Figure 1: Rappel UML du patron décorateur.

2 décorations concrètes : chocolat ou chantilly

```
class Dessert {
 public :
  virtual string getName() const =0;
  virtual double getPrice() const =0;
};

class Gaufre : public Dessert {
 public :
  string getName() const { return "gaufre";}
  double getPrice() const {return 1.8;}
};

class Crepe : public Dessert {
 public :
  string getName() const { return "crepe";}
  double getPrice() const {return 1.5;}
};
```

```
class Deco : public Dessert {
 protected : // sera utile aux sous classes
   const Dessert & ref;
public :
     Deco (const Dessert & x) : ref{x} {}
};
```

```
class Deco : public Dessert {
 protected : // sera utile aux sous classes
   const Dessert & ref;
public :
     Deco (const Dessert & x) : ref{x} {}
};
```

```
class Choco : public Deco {
 public :
   Choco(const Dessert & x): Deco{x} {}
   string getName() const { return ref.getName()+ " au
chocolat ";}
   double getPrice() const {return ref.getPrice()+0.2;}
};

class Chanti : public Deco {
 public :
   Chanti(const Dessert & x): Deco{x} {}
   string getName() const { return ref.getName()+ " à la
chantilly ";}
   double getPrice() const {return ref.getPrice()+0.5;}
};
```

```
int main() {
    Dessert &d { Choco(Chanti(Choco(Crepe{}))) } ;
    cout << d.getName() << d.getPrice() << endl;
     return EXIT_SUCCESS;
}
```