

CORRECTION D'EXAMEN

2022 – 2023

Session 2 ...

CORRECTION D'EXAMEN



13/06/2023 - 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.**

Mêmes consignes :

- se concentrer à bien faire 2/3 du sujet, se relire.
- vous pourrez « courir après le score » ensuite.

Exercice 1 (4 points) L'objectif de cet exercice est de manipuler des fenêtres (classe Window) auxquelles on peut ajouter une barre de titre (classe Titlebar) et/ou un ascenseur (classe Slider). Sur chacun de ces objets on peut invoquer une méthode void draw() le dessinant. Dans l'exercice, ces méthodes se contentent d'afficher un texte : la méthode draw d'une fenêtre affiche un texte et ses dimensions, celles des objets Titlebar et Slider affichent un texte puis délèguent le reste à un objet référencé en interne.

Pour que vos réponses soient lisibles sur papier :

- commencez au brouillon afin de faire vos ajustements sans ratures
- par soucis de lisibilité ne séparez pas les .hpp et .cpp : écrivez tout dans un .cpp par classe.
- ne vous préoccupez pas des copies, ni des destructions, ce qui nous intéresse est la structure que vous concevez pour que le code fasse ce qui est indiqué.

Question : écrivez les classes Window, Titlebar et Slider en ajoutant éventuellement d'autres classes pour que le code ci-dessous fonctionne et donne les affichages mis en commentaire. Vous pouvez également donner quelques indications sur le raisonnement qui à conduit à vos choix.

```
int main() {
2  Window* win0 = new Window(10, 10);
   win0->draw(); // Je dessine une fenetre 10x10
4
   Slider* win1 = new Slider(new Window(10, 20));
6  win1->draw();
   // Je dessine l'ascenseur
8  // Je dessine une fenetre 10x20

10 Titlebar* win2 = new Titlebar("Win2", new Window(20, 10));
   win2->draw();
12 // Je dessine la barre de titre Win2
   // Je dessine une fenetre 20x10
14
   Titlebar* win3 = new Titlebar("Win3", new Slider(new Window(20, 20)));
16 win3->draw();
   // Je dessine la barre de titre Win3
18 // Je dessine l'ascenseur
   // Je dessine une fenetre 20x20
20

22 Slider* win4 = new Slider(new Titlebar("Win4", new Window(20, 20)));
   win4->draw();
24 // Je dessine l'ascenseur
   // Je dessine la barre de titre Win4
26 // Je dessine une fenetre 20x20
}
```

Exercice 1 (4 points) L'objectif de cet exercice est de manipuler des fenêtres (classe Window) auxquelles c'est à dire : la hiérarchie des classes, l'inclusion d'objets, la méthode draw (classe Slider). Sur chacun de ces exercices, ces méthodes se contentent de manipuler les dimensions, celles des objets référencé en interne.

- commencez au brouillon afin de faire vos essais sans ratures
- par soucis de lisibilité ne séparez pas les .hpp et .cpp : écrivez tout dans un .cpp par classe.
- ne vous préoccupez pas des copies, ni des destructions, ce qui nous intéresse est la structure que vous concevez pour que le code fasse ce qui est indiqué.

Question : écrivez les classes Window, Titlebar et Slider en ajoutant éventuellement d'autres classes pour que le code ci-dessous fonctionne et donne les affichages mis en commentaire. Vous pouvez également donner quelques indications sur le raisonnement qui a conduit à vos choix.

```
int main() {
2  Window* win0 = new Window(10, 10);
   win0->draw(); // Je dessine une fenetre 10x10
4
   Slider* win1 = new Slider(new Window(10, 20));
6  win1->draw();
   // Je dessine l'ascenseur
8  // Je dessine une fenetre 10x20

10 Titlebar* win2 = new Titlebar("Win2", new Window(20, 10));
   win2->draw();
12 // Je dessine la barre de titre Win2
   // Je dessine une fenetre 20x10
14

16 Titlebar* win3 = new Titlebar("Win3", new Slider(new Window(20, 20)));
   win3->draw();
   // Je dessine la barre de titre Win3
18 // Je dessine l'ascenseur
   // Je dessine une fenetre 20x20
20

22 Slider* win4 = new Slider(new Titlebar("Win4", new Window(20, 20)));
   win4->draw();
24 // Je dessine l'ascenseur
   // Je dessine la barre de titre Win4
26 // Je dessine une fenetre 20x20
}
```

Exercice 1 (4 points) L'objectif de cet exercice est de manipuler des fenêtres (classe Window) auxquelles on peut ajouter une barre de titre (classe Titlebar) et/ou un ascenseur (classe Slider). Sur chacun de ces objets on peut invoquer une méthode void draw() le dessinant. Dans l'exercice, ces méthodes se contentent d'afficher un texte : la méthode draw d'une fenêtre affiche un texte et ses dimensions, celles des objets Titlebar et Slider affichent un texte puis délèguent le reste à un objet référencé en interne.

Pour que vos réponses soient lisibles sur papier :

- commencez au brouillon afin de faire vos ajustements sans ratures
- par soucis de lisibilité ne séparez pas les .hpp et .cpp
- ne vous préoccupez pas des copies, ni des descripteurs de fichiers, vous concevez pour que le code fasse ce qui est demandé.

Question : écrivez les classes Window, Titlebar et Slider pour que le code ci-dessous fonctionne et donne les affichages indiqués. Vous pouvez également donner quelques indications sur le raisonnement qui a conduit à votre choix.

```
1 int main() {
2   Window* win0 = new Window(10, 10);
3   win0->draw(); // Je dessine une fenetre 10x10
4
5   Slider* win1 = new Slider(new Window(10, 20));
6   win1->draw();
7   // Je dessine l'ascenseur
8   // Je dessine une fenetre 10x20
9
10  Titlebar* win2 = new Titlebar("Win2", new Window(20, 10));
11  win2->draw();
12  // Je dessine la barre de titre Win2
13  // Je dessine une fenetre 20x10
14
15  Titlebar* win3 = new Titlebar("Win3", new Slider(new Window(20, 20)));
16  win3->draw();
17  // Je dessine la barre de titre Win3
18  // Je dessine l'ascenseur
19  // Je dessine une fenetre 20x20
20
21
22  Slider* win4 = new Slider(new Titlebar("Win4", new Window(20, 20)));
23  win4->draw();
24  // Je dessine l'ascenseur
25  // Je dessine la barre de titre Win4
26  // Je dessine une fenetre 20x20
27 }
```

Slider est construit
« autour » d'une Window

Exercice 1 (4 points) L'objectif de cet exercice est de manipuler des fenêtres (classe Window) auxquelles on peut ajouter une barre de titre (classe Titlebar) et/ou un ascenseur (classe Slider). Sur chacun de ces objets on peut invoquer une méthode void draw() le dessinant. Dans l'exercice, ces méthodes se contentent d'afficher un texte : la méthode draw d'une fenêtre affiche un texte et ses dimensions, celles des objets Titlebar et Slider affichent un texte puis délèguent le reste à un objet référencé en interne.

Pour que vos réponses soient lisibles sur papier :

- commencez au brouillon afin de faire vos ajustements sans ratures
- par soucis de lisibilité ne séparez pas les .hpp et .cpp
- ne vous préoccupez pas des copies, ni des descripteurs de fichiers, vous concevez pour que le code fasse ce qui est demandé

Question : écrivez les classes Window, Titlebar et Slider pour que le code ci-dessous fonctionne et donne les affichages attendus. Vous pouvez également donner quelques indications sur le raisonnement qui a conduit à votre solution.

```
1 int main() {
2   Window* win0 = new Window(10, 10);
3   win0->draw(); // Je dessine une fenetre 10x10
4
5   Slider* win1 = new Slider(new Window(10, 20));
6   win1->draw();
7   // Je dessine l'ascenseur
8   // Je dessine une fenetre 10x20
9
10  Titlebar* win2 = new Titlebar("Win2", new Window(20, 10));
11  win2->draw();
12  // Je dessine la barre de titre Win2
13  // Je dessine une fenetre 20x10
14
15  Titlebar* win3 = new Titlebar("Win3", new Slider(new Window(20, 20)));
16  win3->draw();
17  // Je dessine la barre de titre Win3
18  // Je dessine l'ascenseur
19  // Je dessine une fenetre 20x20
20
21
22  Slider* win4 = new Slider(new Titlebar("Win4", new Window(20, 20)));
23  win4->draw();
24  // Je dessine l'ascenseur
25  // Je dessine la barre de titre Win4
26  // Je dessine une fenetre 20x20
27 }
```

Slider est construit
« autour » d'une Window
ou d'une Titlebar

Exercice 1 (4 points) L'objectif de cet exercice est de manipuler des fenêtres (classe Window) auxquelles on peut ajouter une barre de titre (classe Titlebar) et/ou un ascenseur (classe Slider). Sur chacun de ces objets on peut invoquer une méthode void draw() le dessinant. Dans l'exercice, ces méthodes se contentent d'afficher un texte : la méthode draw d'une fenêtre affiche un texte et ses dimensions, celles des objets Titlebar et Slider affichent un texte puis délèguent le reste à un objet référencé en interne.

Pour que vos réponses soient lisibles sur papier :

- commencez au brouillon afin de faire vos ajustements sans ratures
- par soucis de lisibilité ne séparez pas les .hpp et .cpp
- ne vous préoccupez pas des copies, ni des descripteurs de fichiers, vous concevez pour que le code fasse ce qui est demandé

Question : écrivez les classes Window, Titlebar et Slider pour que le code ci-dessous fonctionne et donne les affichages indiqués. Vous pouvez également donner quelques indications sur le raisonnement qui a conduit à votre solution.

```
1 int main() {
2   Window* win0 = new Window(10, 10);
3   win0->draw(); // Je dessine une fenetre 10x10
4
5   Slider* win1 = new Slider(new Window(10, 20));
6   win1->draw();
7   // Je dessine l'ascenseur
8   // Je dessine une fenetre 10x20
9
10  Titlebar* win2 = new Titlebar("Win2", new Window(20, 10));
11  win2->draw();
12  // Je dessine la barre de titre Win2
13  // Je dessine une fenetre 20x10
14
15  Titlebar* win3 = new Titlebar("Win3", new Slider(new Window(20, 20)));
16  win3->draw();
17  // Je dessine la barre de titre Win3
18  // Je dessine l'ascenseur
19  // Je dessine une fenetre 20x20
20
21
22  Slider* win4 = new Slider(new Titlebar("Win4", new Window(20, 20)));
23  win4->draw();
24  // Je dessine l'ascenseur
25  // Je dessine la barre de titre Win4
26  // Je dessine une fenetre 20x20
27 }
```

Titlebar est construit
« autour » d'une Window

Exercice 1 (4 points) L'objectif de cet exercice est de manipuler des fenêtres (classe Window) auxquelles on peut ajouter une barre de titre (classe Titlebar) et/ou un ascenseur (classe Slider). Sur chacun de ces objets on peut invoquer une méthode void draw() le dessinant. Dans l'exercice, ces méthodes se contentent d'afficher un texte : la méthode draw d'une fenêtre affiche un texte et ses dimensions, celles des objets Titlebar et Slider affichent un texte puis délèguent le reste à un objet référencé en interne.

Pour que vos réponses soient lisibles sur papier :

- commencez au brouillon afin de faire vos ajustements sans ratures
- par soucis de lisibilité ne séparez pas les .hpp et .cpp
- ne vous préoccupez pas des copies, ni des descripteurs de fichiers, vous concevez pour que le code fasse ce qui est demandé

Question : écrivez les classes Window, Titlebar et Slider pour que le code ci-dessous fonctionne et donne les affichages attendus. Vous pouvez également donner quelques indications sur le raisonnement qui a conduit à votre solution.

```
1 int main() {
2   Window* win0 = new Window(10, 10);
3   win0->draw(); // Je dessine une fenetre 10x10
4
5   Slider* win1 = new Slider(new Window(10, 20));
6   win1->draw();
7   // Je dessine l'ascenseur
8   // Je dessine une fenetre 10x20
9
10  Titlebar* win2 = new Titlebar("Win2", new Window(20, 10));
11  win2->draw();
12  // Je dessine la barre de titre Win2
13  // Je dessine une fenetre 20x10
14
15  Titlebar* win3 = new Titlebar("Win3", new Slider(new Window(20, 20)));
16  win3->draw();
17  // Je dessine la barre de titre Win3
18  // Je dessine l'ascenseur
19  // Je dessine une fenetre 20x20
20
21
22  Slider* win4 = new Slider(new Titlebar("Win4", new Window(20, 20)));
23  win4->draw();
24  // Je dessine l'ascenseur
25  // Je dessine la barre de titre Win4
26  // Je dessine une fenetre 20x20
27 }
```

Titlebar est construit
« autour » d'une Window
ou d'un Slider

Exercice 1 (4 points) L'objectif de cet exercice est de manipuler des fenêtres (classe Window) auxquelles on peut ajouter une barre de titre (classe Titlebar) et/ou un ascenseur (classe Slider). Sur chacun de ces objets on peut invoquer une méthode void draw() le dessinant. Dans l'exercice, ces méthodes se contentent d'afficher un texte : la méthode draw d'une fenêtre affiche un texte et ses dimensions, celles des objets Titlebar et Slider affichent un texte puis délèguent le reste à un objet référencé en interne.

Pour que vos réponses soient lisibles sur papier :

- commencez au brouillon afin de faire vos ajustements sans ratures
- par soucis de lisibilité ne séparez pas les .hpp et .cpp
- ne vous préoccupez pas des copies, ni des descripteurs de fichiers, vous concevez pour que le code fasse ce qui est demandé

Question : écrivez les classes Window, Titlebar et Slider pour que le code ci-dessous fonctionne et donne les affichages attendus. Vous pouvez également donner quelques indications sur le raisonnement qui a conduit à votre solution.

```
1 int main() {
2   Window* win0 = new Window(10, 10);
3   win0->draw(); // Je dessine une fenetre 10x10
4
5   Slider* win1 = new Slider(new Window(10, 20));
6   win1->draw();
7   // Je dessine l'ascenseur
8   // Je dessine une fenetre 10x20
9
10  Titlebar* win2 = new Titlebar("Win2", new Window(20, 10));
11  win2->draw();
12  // Je dessine la barre de titre Win2
13  // Je dessine une fenetre 20x10
14
15  Titlebar* win3 = new Titlebar("Win3", new Slider(new Window(20, 20)));
16  win3->draw();
17  // Je dessine la barre de titre Win3
18  // Je dessine l'ascenseur
19  // Je dessine une fenetre 20x20
20
21
22  Slider* win4 = new Slider(new Titlebar("Win4", new Window(20, 20)));
23  win4->draw();
24  // Je dessine l'ascenseur
25  // Je dessine la barre de titre Win4
26  // Je dessine une fenetre 20x20
27 }
```

Titlebar est construit
« autour » d'une Window
ou d'un Slider

Tous ont un type commun,
disons Graphics

```
class Graphics {  
public:  
    virtual void draw() const =0 ;  
};
```

```
class Graphics {  
public:  
    virtual void draw() const =0 ;  
};
```

```
class Window : public Graphics {  
public:  
    Window(int a, int b): x{a},y{b} {}  
    void draw() const {  
        cout    <<  "Je dessine une fenetre "  
                <<  x << "*" << y << endl;  
    }  
private: int x,y;  
};
```

```
class Graphics {
public:
    virtual void draw() const =0 ;
};
```

```
class Window : public Graphics {
public:
    Window(int a, int b): x{a},y{b} {}
    void draw() const {
        cout << "Je dessine une fenetre "
              << x << "x" << y << endl;
    }
private: int x,y;
};
```

```
class Slider : public Graphics {
public:
    Slider(Graphics * x):w{x} {};
    void draw() const {
        cout << "Je dessine l'ascenseur" << endl;
        w-> draw();
    };
private: Graphics * w;
};
```



```
class Titlebar : public Graphics {
public:
    Titlebar(string n, Graphics * w ):name{n},content {w}{}
    void draw() const {
        cout    << "Je dessine la barre de titre "
                << name << endl;
        content->draw();
    }

private:
    string name;
    Graphics * content;
};
```

Exercice 2 (6 points)

- Le code suivant modélise des listes simplement chaînées.

```
1 class Noeud {  
    public :  
3     int valeur;  
     Noeud* suivant;  
5     Noeud (int v, Noeud *n):valeur{v}, suivant{n} {}  
     ~Noeud() {}  
7 };  
class ListeChaine {  
9     private:  
     Noeud* tete;  
11    public:  
     ListeChaine() : tete{nullptr} {}  
13     void ajouter(int valeur) { tete = new Noeud (valeur, tete); }  
     // Destructeur  
15     ~ListeChaine() { .... // a completer }  
};
```

1. Ecrivez le destructeur de listes.
2. En réfléchissant un peu à l'utilisation habituelle qu'on fait des listes (ajout, suppression, opérations de parcours ou autre) j'aimerais que vous trouviez une justification (en quelques lignes) du fait que le destructeur de noeud est bien celui qui est écrit dans ce code.

Exercice 2 (6 points)

- Le code suivant modélise des listes simplement chaînées.

```
1 class Noeud {  
  public :  
3   int valeur;  
   Noeud* suivant;  
5   Noeud (int v, Noeud *n):valeur{v}, suivant{n} {}  
   ~Noeud() {}  
7 };  
class ListeChaine {  
9   private:  
   Noeud* tete;  
11  public:  
   ListeChaine() : tete{nullptr} {}  
13   void ajouter(int valeur) { tete = new Noeud (valeur, tete); }  
   // Destructeur  
15   ~ListeChaine() { .... // a completer }  
};
```

on remarque que la liste
est responsable de la
création de Noeuds

- Ecrivez le destructeur de listes.
- En réfléchissant un peu à l'utilisation habituelle qu'on fait des listes (ajout, suppression, opérations de parcours ou autre) j'aimerais que vous trouviez une justification (en quelques lignes) du fait que le destructeur de noeud est bien celui qui est écrit dans ce code.

Exercice 2 (6 points)

- Le code suivant modélise des listes simplement chaînées.

```
1 class Noeud {
   public :
3     int valeur;
     Noeud* suivant;
5     Noeud (int v, Noeud *n):valeur{v}, suivant{n} {}
     ~Noeud() {}
7 };
class ListeChaine {
9     private:
     Noeud* tete;
11    public:
     ListeChaine() : tete(nullptr) {}
13    void ajouter(int va) {
        // Destructeur
15    ~ListeChaine() {
        }
};

~ListeChaine() {
    Noeud* courant = tete;
    while (courant != nullptr) {
        Noeud* suivant = courant->suivant;
        delete courant;
        courant = suivant;
    }
}
```

- Ecrivez le destructeur de listes.
- En réfléchissant un peu à l'utilisation habituelle qu'on fait des listes (ajout, suppression, opérations de parcours ou autre) j'aimerais que vous trouviez une justification (en quelques lignes) du fait que le destructeur de noeud est bien celui qui est écrit dans ce code.

- Ecrivez le destructeur qui convient pour les vecteurs d'entiers :

```
1 class VecteurEntier {  
    private:  
3     int* elements; // Pointeur vers le tableau d'entiers  
     int taille;    // Taille du tableau  
5     public:  
     VecteurEntier(int taille) : taille{taille}, elements {new int[taille]} {}  
7     // Destructeur  
     ~VecteurEntier() { .... // a completer }  
9 }
```

```
~VecteurEntier() {  
    delete [] elements ;  
}
```

- Quels sont les affichages produits lors de l'exécution du main suivant ? (Expliquez un peu mais clairement votre réponse)

```
1  class A {  
2      public :  
3          virtual ~A() { cout << "del A" << endl;}  
4  };  
  
6  class B : public A {  
7      public :  
8          virtual ~B(){ cout << "del B" << endl;}  
9  };  
10  
11  class C {  
12      public :  
13          ~C() { cout << "del C" << endl;}  
14  };  
  
16  class D : public C {  
17      public :  
18          virtual ~D(){ cout << "del D" << endl;}  
19  };  
20  
21  int main() {  
22      A * a = new B();  
23      C * c = new D();  
24      delete a;  
25      delete c;  
26  }
```

- Quels sont les affichages produits lors de l'exécution du main suivant ? (Expliquez un peu mais clairement votre réponse)

```
1 class A {  
2     public :  
3         virtual ~A() { cout << "del A" << endl; }  
4 };  
  
5  
6 class B : public A {  
7     public :  
8         virtual ~B() { cout << "del B" << endl; }  
9 };  
10  
11 class C {  
12     public :  
13         ~C() { cout << "del C" << endl; }  
14 };
```

Il faut noter ici que
ce destructeur n'est
pas virtual

```
15  
16 class D : public C {  
17     public :  
18         virtual ~D() { cout << "del D" << endl; }  
19 };  
20  
21  
22 int main() {  
23     A * a = new B();  
24     C * c = new D();  
25     delete a;  
26     delete c;  
27 }
```

- Quels sont les affichages produits lors de l'exécution du main suivant ? (Expliquez un peu mais clairement votre réponse)

```
1 class A {  
2     public :  
3         virtual ~A() { cout << "del A" << endl; }  
4 };  
  
5  
6 class B : public A {  
7     public :  
8         virtual ~B() { cout << "del B" << endl; }  
9 };  
10  
11 class C {  
12     public :  
13         ~C() { cout << "del C" << endl; }  
14 };
```

Il faut noter ici que
ce destructeur n'est
pas virtual

```
15  
16 class D : public C {  
17     public :  
18         virtual ~D() { cout << "del D" << endl; }  
19 };  
20  
21  
22 int main() {  
23     A * a = new B();  
24     C * c = new D();  
25     delete a;  
26     delete c;  
27 }
```

```
del B  
del A  
del C
```


Exercice 3 (4 points) On cherche à caractériser les animaux en relevant leur couleur et leur poids dans des attributs `private`. On établit ensuite une classification principalement en les séparant entre les sous-classes des oiseaux et des mammifères. Puis on complètera éventuellement (ne le faites pas) cette hiérarchie par des races (Chien, Poule, etc ...). Ce qui nous intéresse vraiment est de traiter un cas particulier : celui de la chauve-souris qui a la particularité de voler et d'être un mammifère ...

```
class Animal {  
2 public:  
    void presentation() {  
4        whatIam(); // a faire ...  
        cout << "de couleur " << couleur << " et de poids " << poids << endl;  
6    }  
    Animal(string c, int p): couleur{c}, poids{p}{}  
8  
private :  
10    string couleur;  
    int poids;  
12 }
```

1. Ecrivez la hiérarchie de classes la plus adaptée qui permet de modéliser les chauve-souris. (On veut voir apparaître le code des classes en se focalisant sur l'héritage, et que vous écriviez un constructeur simple par classe)
2. La méthode `whatIam()` **n'est pas publique**. Ecrivez là, de sorte que la présentation d'un exemple trivial composé d'un animal, d'un mammifère, d'un oiseau et d'une chauve souris donne :

```
je suis un animal inconnu de couleur blanc et de poids 100  
2 je suis un mammifere de couleur rose et de poids 200  
je suis un oiseau de couleur vert et de poids 300  
4 je suis un mammifere je suis un oiseau de couleur noire et de poids 400
```

Merci de respecter cette consigne : en particulier une chauve souris ne dit pas qu'elle est une chauve souris, et chaque animal ne donne son poids et sa couleur qu'une fois.

```
class Mammifere : virtual public Animal {
public:
    Mammifere(string c, int p): Animal(c,p) {}
}
class Oiseau : virtual public Animal {
public:
    Oiseau (string c, int p): Animal{c,p} {}
}
class ChauveSouris : public Mammifere, public Oiseau {
public:
    ChauveSouris(string c, int p): Animal{c,p},
                                   Mammifere{c,p}, Oiseau{c,p}, {}
}
```

On rappelle que l'héritage indiqué comme virtual dans une hierarchie "en diamant" assure qu'un seul Animal soit créé malgré les 2 branches.

En complément, cet Animal doit être créé "au plus prêt" de la classe concrètes, c'est à dire au niveau de la Chauve-Souris.

Les constructions d'animaux via les autres constructeurs sont ensuite court-circuitées

Exercice 3 (4 points) On cherche à caractériser les animaux par des attributs `private`. On établit ensuite une hiérarchie de sous-classes des oiseaux et des mammifères. Pour la hiérarchie par des races (Chien, Poule, etc ...), on a un cas particulier : celui de la chauve-souris qui a la particularité d'être un mammifère et un oiseau.

Elle sera donc soit `private`, soit `protected`.

```
1 class Animal {
2     public:
3         void presentation() {
4             whatIam(); // a faire ...
5             cout << "de couleur " << couleur << " et de poids " << poids << endl;
6         }
7         Animal(string c, int p): couleur{c}, poids{p} {}
8
9     private :
10        string couleur;
11        int poids;
12 }
```

1. Ecrivez la hiérarchie de classes la plus adaptée qui permet de modéliser les chauve-souris. (On veut voir apparaître le code des classes en se focalisant sur l'héritage, et que vous écriviez un constructeur simple par classe)
2. La méthode `whatIam()` **n'est pas publique**. Ecrivez là, de sorte que la présentation d'un exemple trivial composé d'un animal, d'un mammifère, d'un oiseau et d'une chauve souris donne :

```
1 je suis un animal inconnu de couleur blanc et de poids 100
2 je suis un mammifere de couleur rose et de poids 200
3 je suis un oiseau de couleur vert et de poids 300
4 je suis un mammifere je suis un oiseau de couleur noire et de poids 400
```

Merci de respecter cette consigne : en particulier une chauve souris ne dit pas qu'elle est une chauve souris, et chaque animal ne donne son poids et sa couleur qu'une fois.

Exercice 3 (4 points) On cherche à caractériser les animaux par des attributs `private`. On établit ensuite une hiérarchie de sous-classes des oiseaux et des mammifères. Pour la hiérarchie par des races (Chien, Poule, etc ...), on a un cas particulier : celui de la chauve-souris qui a la particularité d'être un oiseau et un mammifère.

Remarquez que `presentation` n'est pas `virtual`

```
1 class Animal {
2 public:
3     void presentation() {
4         whatIam(); // a faire ...
5         cout << "de couleur " << couleur << " et de poids " << poids << endl;
6     }
7     Animal(string c, int p): couleur{c}, poids{p}{}
8
9 private :
10     string couleur;
11     int poids;
12 }
```

1. Ecrivez la hiérarchie de classes la plus adaptée qui permet de modéliser les chauve-souris. (On veut voir apparaître le code des classes en se focalisant sur l'héritage, et que vous écriviez un constructeur simple par classe)
2. La méthode `whatIam()` **n'est pas publique**. Ecrivez là, de sorte que la présentation d'un exemple trivial composé d'un animal, d'un mammifère, d'un oiseau et d'une chauve souris donne :

```
1 je suis un animal inconnu de couleur blanc et de poids 100
2 je suis un mammifere de couleur rose et de poids 200
3 je suis un oiseau de couleur vert et de poids 300
4 je suis un mammifere je suis un oiseau de couleur noire et de poids 400
```

Merci de respecter cette consigne : en particulier une chauve souris ne dit pas qu'elle est une chauve souris, et chaque animal ne donne son poids et sa couleur qu'une fois.

whatIam doit s'adapter, elle sera donc virtual.

Exercice 3 (4 points) On cherche à caractériser les animaux par des attributs private. On établit ensuite une hiérarchie de sous classes des oiseaux et des mammifères. Pour la hiérarchie par des races (Chien, Poule, etc ...) on a un cas particulier : celui de la chauve-souris qui a la particularité d'être un

```
1 class Animal {
2     public:
3         void presentation() {
4             whatIam(); // a faire ...
5             cout << "de couleur " << couleur << " et de poids " << poids << endl;
6         }
7         Animal(string c, int p): couleur{c}, poids{p}{}
8
9     private :
10        string couleur;
11        int poids;
12 }
```

1. Ecrivez la hiérarchie de classes la plus adaptée qui permet de modéliser les chauve-souris. (On veut voir apparaître le code des classes en se focalisant sur l'héritage, et que vous écriviez un constructeur simple par classe)
2. La méthode `whatIam()` **n'est pas publique**. Ecrivez là, de sorte que la présentation d'un exemple trivial composé d'un animal, d'un mammifère, d'un oiseau et d'une chauve souris donne :

```
1 je suis un animal inconnu de couleur blanc et de poids 100
2 je suis un mammifere de couleur rose et de poids 200
3 je suis un oiseau de couleur vert et de poids 300
4 je suis un mammifere je suis un oiseau de couleur noire et de poids 400
```

Merci de respecter cette consigne : en particulier une chauve souris ne dit pas qu'elle est une chauve souris, et chaque animal ne donne son poids et sa couleur qu'une fois.

Exercice 3 (4 points) On cherche à caractériser les animaux par des attributs `private`. On établit ensuite une hiérarchie de sous-classes des oiseaux et des mammifères. Pour la hiérarchie par des races (Chien, Poule, etc ...), on a un cas particulier : celui de la chauve-souris qui a la particularité d'être un mammifère et un oiseau.

whatIam doit s'adapter, elle sera donc `virtual`.
Visible dans `Animal`, donc `protected` et pas `private`

```
1 class Animal {
2     public:
3         void presentation() {
4             whatIam(); // a faire ...
5             cout << "de couleur " << couleur << " et de poids " << poids << endl;
6         }
7         Animal(string c, int p): couleur{c}, poids{p}{}
8
9     private :
10        string couleur;
11        int poids;
12 }
```

1. Ecrivez la hiérarchie de classes la plus adaptée qui permet de modéliser les chauve-souris. (On veut voir apparaître le code des classes en se focalisant sur l'héritage, et que vous écriviez un constructeur simple par classe)
2. La méthode `whatIam()` **n'est pas publique**. Ecrivez là, de sorte que la présentation d'un exemple trivial composé d'un animal, d'un mammifère, d'un oiseau et d'une chauve souris donne :

```
1 je suis un animal inconnu de couleur blanc et de poids 100
2 je suis un mammifere de couleur rose et de poids 200
3 je suis un oiseau de couleur vert et de poids 300
4 je suis un mammifere je suis un oiseau de couleur noire et de poids 400
```

Merci de respecter cette consigne : en particulier une chauve souris ne dit pas qu'elle est une chauve souris, et chaque animal ne donne son poids et sa couleur qu'une fois.

Exercice 3 (4 points) On cherche à caractériser les animaux en relevant leur couleur et leur poids dans des attributs private. On établit ensuite une classification principalement en les séparant entre les sous classes des oiseaux et des mammifères. Puis on complètera éventuellement (ne le faites pas) cette hiérarchie par des races (Chien, Poule, etc ...). Ce qui nous intéresse vraiment est de traiter un cas particulier : celui de la chauve-souris qui a la particularité de voler et d'être un mammifère ...

```
class Animal {
2 public:
    void presentation() {
4        whatIam(); // a faire ...
        cout << "de couleur " << couleur << endl;
6    }
    Animal(string couleur, int poids) : couleur(couleur), poids(poids) {}
8 private:
10     string couleur;
12     int poids;
}

Animal * x1 = new Animal("blanc", 100);
Animal * x2 = new Mammifere("rose", 200);
Animal * x3 = new Oiseau("vert", 300);
Animal * x4 = new ChauveSouris("noire", 400);

x1 -> presentation();
x2 -> presentation();
x3 -> presentation();
x4 -> presentation();
```

1. Ecrivez la hiérarchie

voir apparaître le code des classes en se focalisant sur l'héritage, et que vous écriviez un constructeur simple par classe)

2. La méthode `whatIam()` **n'est pas publique**. Ecrivez là, de sorte que la présentation d'un exemple trivial composé d'un animal, d'un mammifère, d'un oiseau et d'une chauve souris donne :

```
je suis un animal inconnu de couleur blanc et de poids 100
2 je suis un mammifere de couleur rose et de poids 200
je suis un oiseau de couleur vert et de poids 300
4 je suis un mammifere je suis un oiseau de couleur noire et de poids 400
```

Merci de respecter cette consigne : en particulier une chauve souris ne dit pas qu'elle est une chauve souris, et chaque animal ne donne son poids et sa couleur qu'une fois.

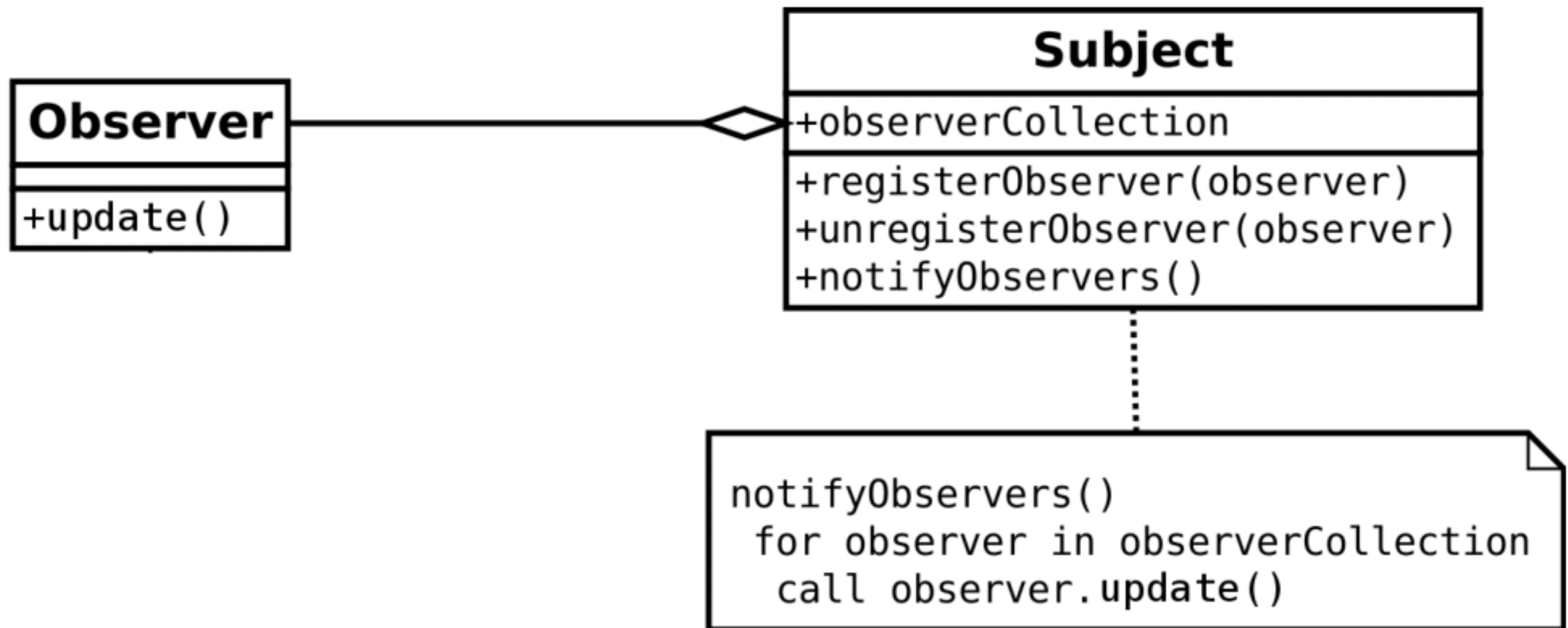
```
class Animal {
    protected : virtual void whatIam() {
        cout << "Je suis un animal inconnu. ";
    }
}

class Mammifere : virtual public Animal {
    protected : virtual void whatIam() {
        cout << "Je suis un mammifère. " ;
    }
}

class Oiseau : virtual public Animal {
    protected : virtual void whatIam() {
        cout << "Je suis un oiseau. " ;
    }
}

class ChauveSouris : public Mammifere, public Oiseau {
    protected : virtual void whatIam() {
        Mammifere::whatIam();
        Oiseau::whatIam();
    }
}
```


Exercice 4 (6 points) On rappelle que le patron de conception "Observateur" est utilisé lorsque des objets, appelés sujets, maintiennent une liste d'objets dépendants, appelés observateurs, et les informent automatiquement de tout changement d'état. Vous trouverez son diagramme ci-dessous. Notez bien que les signatures sont indicatives, dans la mesure où la communication entre sujet et observateur peut être enrichie en transmettant des données utiles.



Remarquez qu'il faut interpréter un peu ce diagramme et l'adapter au cadre de c++ :

ajuster le type de la collection

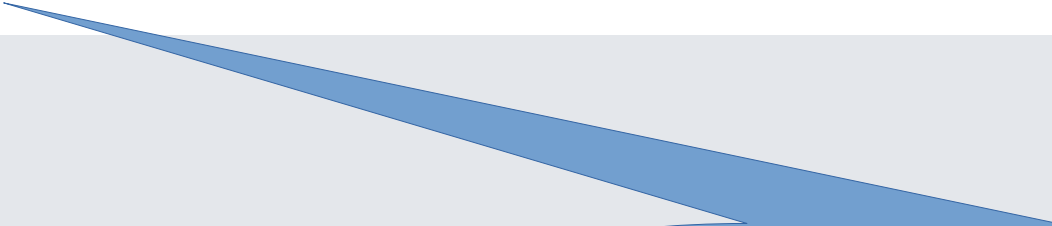
ajuster le type des arguments pour register/unregister : les objets préexistent, on utilisera probablement des alias (références)

Nous allons nous intéresser à un contexte qui est celui de la diffusion de vidéos en streaming entre ce que nous appellerons des Serveur (canal plus, disney, netflix), et des postes de visionnage qui seront leurs Client.

1. Dans ce cas, qui joue le rôle de l'Observer ? et qui joue celui du Subject ?

Nous allons nous intéresser à un contexte qui est celui de la diffusion de vidéos en streaming entre ce que nous appellerons des Serveur (canal plus, disney, netflix), et des postes de visionnage qui seront leurs Client.

1. Dans ce cas, qui joue le rôle de l'Observer ? et qui joue celui du Subject ?



Observer : les Clients
Subject : les Serveurs

. Voici quelques éléments préalables à l'écriture des classes :

- (a) Seule l'image du direct sera stockée côté serveur (nous supposons pour cela qu'une classe Image est disponible, cela n'a pas d'autre importance). L'observateur, lui, enregistrera tout ce qui l'intéresse dans une liste pour permettre son visionnage ¹. Cela devrait vous donner une première indication sur les attributs à introduire, et sur les signatures les plus adaptées pour les notifications.
- (b) Il faudra prévoir un setter de l'image du direct (...)
- (c) L'observateur devra être construit avec en paramètre une référence vers son sujet, cela aura une conséquence **à prendre en compte lors de la destruction de l'observateur** ; ainsi que sur **l'ordre d'écriture de votre code**.
- (d) L'enregistrement, et la désinscription sont purement déclaratives, elles se feront sans contraintes (en réalité il faudrait appliquer la politique commerciale).
- (e) ne vous préoccupez pas de constructeurs de copies, concentrez vous sur la modélisation du problème faite dans le cadre de ce pattern.

A présent vous devriez être en mesure d'écrire les deux classes Serveur et Client. Un conseil : faites le d'abord au brouillon. Ici, séparez .hpp et .cpp et **dites bien dans quel ordre** vous les écririez dans le compilateur, si vous faites des déclarations préalables etc ... Pour info, ma correction fait une trentaine de lignes en tout, pas plus.

¹Vous pouvez utiliser la librairie standard et "inventer" les opérations si vous ne vous rappelez pas bien de la syntaxe

```
class Client; // déclaration préalable --- dans Serveur.hpp
class Serveur {
private :
    list<Client *> all;
    string direct; // ou une image ...
    void notify(); // je préfère private ...
public :
    void subscribe (Client & c); // une référence
    void unsubscribe (Client & c); // idem
    void set(string d) ; { // demandée par l'énoncé
};
```

```
class Client; // déclaration préalable --- dans Serveur.hpp
class Serveur {
private :
    list<Client *> all;
    string direct; // ou une image ...
    void notify(); // je préfère private ...
public :
    void subscribe (Client & c); // une référence
    void unsubscribe (Client & c); // idem
    void set(string d) ; { // demandée par l'énoncé
};
```

```
// ----- Serveur.cpp -----
#include "Serveur.hpp"
#include "Client.hpp" // car appel à des méthodes de Client
void Serveur::notify() {
    for (Client *x : all) x->update(direct);
}
void Serveur::subscribe ( Client & c ) {
    all.push_front(& c);
}
void Serveur::set(string d) { direct=d; notify(); }
void Serveur::unsubscribe (Client & c) { all.remove(&c); }
```

```
// ----- Client.hpp -----  
#include "Serveur.hpp"  
class Client {  
public:  
    void update(string x);  
    Client (Serveur & s);  
    virtual ~Client();  
private :  
    list<string> stream;  
    Serveur & tv;  
};
```

```
// ----- Client.cpp -----  
#include "Client.hpp"  
#include "Serveur.hpp" // facultatif  
  
void Client::update(string x) {stream.push_back(x);}  
  
Client::Client (Serveur & s):tv{s} {}  
  
Client::~~Client() {tv.unsubscribe(*this);} // y penser !
```