Exercice 1 :

1)

Membre = a un et Héritage = est un.

Pour les accès :

-Héritage : la class mère (A :: a) fusionne avec la class fille (B :: a)

-Membre : encapsulée (A :: T x), accès avec x.a.

2)

Les champs et les chaines d'héritage construisent l'objet résultat dans l'ordre des déclarations. Pas de différence en termes de stockage. Exemple :

```
struct A {...};
struct B:A{x};
sizeof(B) = sizeof(C); // ou C est
struct C {
    A a;
    x y;
};
```

3)

La construction mémoire ce fait avec alignement à chaque fois (basé sur le plus grand **sizeof**(B)).

4)

On peut choisir l'ordre des champs, alors l'ordre d'héritage est imposer.

Exercice 2 :

1)

```
class A1 {
private:
    int a1;
public:
    A1():a1(0){}
    A1(int v):a1(v){}
    A1(const A1 &v): a1(v.a1){}
    A1 &operator =(const A1 &v){
        if(&v != this){
            a1 = v.a1;
        }
        return *this;
    }
    int get(){
        return a1;
    }
    void set(int v){
        a1 = v;
    }
    ~A1() = default;
};
```

Une class A2 {} est identique.

2)

```
class B : public A1 {
protected:
    A2 v;
};
```

3)

Constructeur par défaut (Cd) de B, lance le Cd de A1 puis Cd de A2.

4)

Code équivalent : B():A1(), v(){}

5)

Constructeur par copie (Cc) de B, lance le Cc de A1 puis le Cc de A2.

6)

Code équivalent : B(const &b): A1(b), v(b.v){}

7)

Assignation par copie (Ac) de B, lance Ac de A1 puis Ac de A2.

8)

Code équivalent :

```
B &operator =(const B &b){
    if(&b != this){
        a1=b.a1;
        v=b.v;
    }
    return this;
}
```

9)

Destructeur par défaut (Dd) de B, lance Dd de A2 puis Dd de A1.

10)

Pas d'écriture équivalente : mécanisme interne.

≈ v.~A2() suivi de a1.~A1(), plus libération de la mémoire sous-jacentes.

11)

Déplacement : par défaut fait des copies champs à champs.

Constructeur par déplacement : constructeur par copie, champ à champ.

Assignation par déplacement : assignation par copie, champ à champ.

Exercice 3 :

1)

a)

```
struct Point2D {
    float x;
    float y;
    Point2D(float u, float v): x(u), y(v) {}
    Point2D(): Point2D(0.f, 0.f) {} //Constructeur délégué
    Point2D(const Point2D &p): x(p.x), y(p.y) {} //Inutile, idem que le
                                                 // constructeur par défaut
    inline void set(float u, float v){
        x = 0;
        y = 0;
    }
};
```

b)

```
class Stroke {
protected:
    Point2D start;
public:
    Stroke(const Point2D &c): start(c) {}
    Stroke() = default;
    Stroke(const Stroke &c) = default;
    ~Stroke() = default;
};
```

c)

Une class abstraite : il n'est pas possible de définir un objet de ce type.

Réponse naïve : non, car si l'objet est abstrait il ne sera jamais copié.

Réponse éclairée : oui, car sera utilisé pour copier la partie Stroke d'un objet concret qui le contient.

d)

```
class Circle : public Stroke {
protected:
    float radius;
public:
    Circle(const Point2D &c, const float r): Stroke(c), radius(r) {}
    Circle(const float r): Stroke(), radius(r) {}
    //Stroke(): inutile mais recommandé
    Circle(const Circle &c): Stroke(c.start), radius(c.radius) {}
    ~Circle() = default;
};
```

e)

```
class Segment : public Stroke {
protected:
    Point2D end;
public:
    Segment(const Segment &s): Stroke(s), end(s.end) {}
    //Ici upcasting de Segment to Stroke
    Segment(const Point2D &p1, const Point2D &p2) : Stroke(p1), end(p2) {}
    ~Segment() = default;
};
```

2)

a)

C'est une méthode virtuelle pure, donc impossible de calculer length() sans connaitre l'objet concret. Elle transforme donc la class en une class abstraite.

b) Dans la class Stroke :

```
virtual float length() = 0;
```

c) Dans la class Circle :

```
virtual float length(){
    return 2.f * 3.14f * radius;
}
```

d)

Dans la class Segment :

```
virtual float length(){
    return dist(start, end);
}
```

Avant la class Segment :

```
inline float sqrf(float x){
    return x*x;
}
inline float dist(const Point2D &p1, const Point2D &p2){
    return sqrtf((sqrf(p1.x-p2.x))+sqrf(p1.y-p2.y));
}
```

3)

a)

Une interface contient que des virtuelle pures.

b)

Juste après la struct Point2D

```
using Vector = Point2D;

class iTransform{
public:
    virtual void Translate(const Vector &t) = 0;
    virtual void Rotate(const float th) = 0;
};
```

c)

On utilisera cette interface en l'héritant et donner les implémentations des méthodes virtuelles dans la class concrète.

d)

Ce n'est pas une bonne idée pour Point2D car :

- Une petite structure : la rendre virtuelle ajoute le pointeur vers la VTABLE, ce qui double la taille de la structure.
- C'est une structure de base, probablement utilisée lors des calculs intensifs (cela va ajouter des indirections).

e) Dans la structure Point2D :

```
void Translate(const Point2D &p){
    x += p.x;
    y += p.y;
}
void Rotate(const float th){
    set(x*cos(th)-y*sin(th),x*sin(th)+y*sin(th));
}
```

f) Dans la class Stroke :

```
class Stroke : public iTransform {
protected:
    Point2D start;
public:
    Stroke(const Point2D &c): start(c) {}
    Stroke() = default;
    Stroke(const Stroke &c) = default;
    virtual float length() = 0;
    void Translate(const Vector &t){
        start.Translate(t);
    }
    ~Stroke() = default;
};
```

g) Dans la class Circle :

```
void Rotate(const float th){

}
```

h) Dans la class Segment :

```
void Translate(const Vector &v){
    start.Translate(v);
    end.Translate(v);
}
void Rotate(const float th){
    start.Rotate(th);
    end.Rotate(th);
}
```

4)

a)

Parce-que operator << est une fonction externe aux class et non une méthode. Donc pas possible d'utiliser le RTTI pour appeler l'affichage sur le type sous-jacent de l'objet.

b)

La méthode virtuelle pure est le seul outil permettant de déterminer automatiquement le type sous-jacent.

c)

En définissant une méthode virtuelle pure intermédiaire.

d) Dans la class Stroke :

```
friend ostream &operator<<(ostream &os, const Stroke *o){
    return o->view(os);
}
```

e) Dans la class Circle :

```
virtual ostream &view(ostream &os)const{
    return os<<"Disk (center = "<<start<<", radius = "<<radius<<")";
}
```

f) Dans la class Segment :

```
virtual ostream &view(ostream &os)const {
    return os << "Segment [" << start << "," << end << "]"
}
```