TESTC++ Sem2 MT & EL 17.05.24 EPFL / R. Boulic

W=blanc, B=bleu, S=saumon, Y=jaune, All = même réponse pour tous

1) (9 pts) Divers usages de `static` :

Le code **ex1.cc** compile sans warning avec -std=c++11 -Wall. Son exécution se termine normalement.

```
1   #include <iostream>
2   using namespace std;
3
4   static int compteur = 100;
5
6   class X
7   {
8         static int compteur;
9         int id;
10  public:
11        X() : id(compteur)
12        {
13              compteur++;
14              cout << "Constructeur X (id = " << id
15                    << "), compteur = "
16                    << compteur << endl;
17        }
18        ~X()
19        {
20              cout << "Destructeur X (id = " << id
21                    << "), compteur = "
22                   << compteur << endl;
23              compteur--;
24        }
25        void afficher() const
26        {
27              cout << "Objet X d'id = " << id
28                    << " | valeur actuelle de compteur = "
29                    << compteur << endl;
30        }
31  };
32
33  int X::compteur(200);
34
35  int main()
36  {
37        compteur = 33;
38        {
39              X x1;
40              X x2;
41              x1.afficher();
42        }
43        X x3;
44        x3.afficher();
45        return 0;
46  }
```

L'exécution de ce code produit plusieurs affichages dans le terminal. Utiliser la page suivante en indiquant dans la colonne de gauche le numéro de la ligne de code causant l'affichage et la colonne de droite pour montrer et JUSTIFIER le texte affiché.

TESTC++ Sem2 MT & EL 17.05.24 EPFL / R. Boulic

W=blanc, B=bleu, S=saumon, Y=jaune, All = même réponse pour tous

N° ligne de code	Texte affiché et JUSTIFICATION [W]
39	Avec le mot clef `static` la classe X définit un attribut de classe `compteur` (ligne 8) qu'elle initialise ligne 33 avec la valeur `200`. Cet attribut appartient à la portée de la classe X c'est pourquoi il masque la variable `static` globale de même nom déclarée ligne 4. C'est donc la valeur de l'attribut de classe qui est manipulée dans les appels des méthodes de la classe, en particulier dans le constructeur de **x1** ligne 39 avec : - D'abord l'initialisation de l'attribut id avec sa valeur initiale de **200** - Puis sa valeur incrémentée d'une unité, **201**, dans l'affichage ligne 16 **`Constructeur X (id = 200), compteur = 201`**
40	Appel du constructeur pour **x2** ligne 40. On utilise la valeur courante **201** de l'attribut de classe pour initialiser son attribut id qui est affiché, puis l'attribut de classe est incrémenté et sa nouvelle valeur **202** est affichée. **`Constructeur X (id = 201), compteur = 202`**
41	Appel de la méthode `afficher` sur l'instance **x1** dont l'attribut id a reçu la valeur **200** à la ligne 39. La valeur courant de l'attribut de classe est **202**. **`Objet X d'id = 200 | valeur actuelle de compteur = 202`**
42	Les instances sont détruites automatiquement à la fin du bloc dans lequel elles ont été déclarées. C'est fait dans l'ordre inverse de leur déclarations, donc d'abord destruction de **x2** avec affichage de son id et de la valeur courante 202 de l'attribut de classe : **`Destructeur X (id = 201), compteur = 202`** L'attribut de classe est décrémenté lors de cet appel, il vaut ensuite **201**. Vient ensuite l'appel du destructeur pour **x1** qui av aussi produire une décrémentation de l'attribut de classe (qui vaudra 200 après l'appel du destructeur): **`Destructeur X (id = 200), compteur = 201`**
43	Même principe que pour la ligne 39, ici pour l'appel du constructeur de x3 pour laquelle l'attribut de classe vaut **200** pour initialiser id avant d'être incrémenté à la valeur **201** et affiché : **`Constructeur X (id = 200), compteur = 201`**
44	Appel de la méthode `afficher` sur l'instance **x3** dont l'attribut **id** a reçu la valeur **200** à la ligne 43. La valeur courant de l'attribut de classe est **201**. **`Objet X d'id = 200 | valeur actuelle de compteur = 201`**
(45[1])46	Appel du destructeur sur l'instance x3 en sortie du bloc (et fin de la fonction main) ; cela affiche son id et la valeur courante de l'attribut qui est 201 avant décrémentation. **`Destructeur X (id = 200), compteur = 201`**

Correspondance des valeurs affichées pour les autres variantes de couleur de copies :

W= blanc	B = bleu	Y = Yellow	S = Salmon
200	33	100	133
201	34	101	134
202	35	102	135

[1] Nous acceptons le numéro de ligne 45 car l'examen original n'avait pas de ligne de numéro 46

W=blanc, B=bleu, S=saumon, Y=jaune, All = même réponse pour tous

2) (9 pts) Héritage : soit le fichier **ex2.cc** dont le code est listé ci-dessous

Ce code **ex2.cc** compile sans warning avec -std=c++11 -Wall. Son exécution se termine normalement.

```
1   #include <iostream>
2   using namespace std;
3
4   class Sensor {
5   public:
6         Sensor( int id ) : id( id ) {
7               cout << "init Sensor : " << id << endl;
8         }
9         virtual ~Sensor() {
10              cout << "destroy Sensor : " << id << endl;
11        }
12        virtual void report() const {
13              cout << "Sensor : " << id << " reporting " << endl;
14        }
15  protected:
16        int id;
17  };
18
19  class TemperatureSensor : public Sensor {
20  public:
21        TemperatureSensor( int id, float temp )
22              : Sensor( id ), temperature( temp ) {
23              cout << "init temperature : "
24                    << temperature << "°C" << endl;
25        }
26        ~TemperatureSensor() {
27              cout << "destroy temperature sensor " << id << endl;
28        }
29        void report() const override {
30              cout << "temperature of " << id << " is "
31                    << temperature << "°C" << endl;
32        }
33  private:
34        float temperature;
35  };
36
37  void sendReport( const Sensor& s ) {
38        s.report();
39  }
40
41  int main() {
42
43        Sensor* s = new TemperatureSensor( 2025, 11.2 );
44
45        s->report();
46
47        sendReport( *s );
48
49        delete s;
50
51        return 0;
52  }
53
```

2.1) JUSTIFIER tous les affichages obtenus par son exécution :

Lignes 43, 21-22, 6-8 : le constructeur de **Sensor** est d'abord appelé :

[W]	`init Sensor : 2025`
[B]	`init Sensor : 2500`
[Y]	`init Sensor : 2250`
[S]	`init Sensor : 2052`

Ligne 23-24 : ensuite seulement vient cet affichage :

[W]	`init temperature : 11.2°C`
[B]	`init temperature : 77.2°C`
[Y]	`init temperature : 17.2°C`
[S]	`init temperature : 71.2°C`

Ligne 43 : L'adresse est mémorisée dans un pointeur de la superclasse

Ligne 45, 29-31 : étant donné que la méthode **report** est **virtual**, et qu'elle est appelée *via un pointeur*, alors c'est la méthode de la classe dérivée qui est substituée qui affiche :

[W]	`temperature of 2025 is 11.2°C`
[B]	`temperature of 2500 is 77.2°C`
[Y]	`temperature of 2250 is 17.2°C`
[S]	`temperature of 2052 is 71.2°C`

Ligne 47, 38, 29-31: étant donné que la méthode **report** est **virtual**, et qu'elle est appelée *via une référence*, alors, ligne 38, c'est la méthode de la classe dérivée qui est substituée et qui affiche :

[W]	`temperature of 2025 is 11.2°C`
[B]	`temperature of 2500 is 77.2°C`
[Y]	`temperature of 2250 is 17.2°C`
[S]	`temperature of 2052 is 71.2°C`

Ligne 49,12-13, 26-28: étant donné que le destructeur de la classe **Sensor** est **virtual** et qu'il est appelée *via un pointeur*, alors c'est le destructeur de la classe dérivée qui est substitué et qui affiche d'abord :

[W]	`destroy temperature sensor 2025`
[B]	`destroy temperature sensor 2500`
[Y]	`destroy temperature sensor 2250`
[S]	`destroy temperature sensor 2052`

Ensuite le destructeur de la classe Sensor est aussi appelé

[W]	`destroy Sensor : 2025`
[B]	`destroy Sensor : 2500`
[Y]	`destroy Sensor : 2250`
[S]	`destroy Sensor : 2052`

2.2) On ajoute un constructeur supplémentaire entre les lignes **25** et **26** avec le code suivant :

```
TemperatureSensor( int id, float temp )
     : id( id ), temperature( temp ) {
     cout << "init temperature : "
          << temperature << "°C" << endl;
}
```

Question : Indiquer si la compilation de ce programme produit toujours un exécutable :
Si vous répondez « oui », préciser s'il y modification de l'affichage obtenu à l'exécution.
Si vous répondez « non », préciser la cause de l'erreur

[All] Non, il y a en fait plusieurs erreurs de compilation. On demande d'en fournir au moins une. Les voici dans l'ordre de détection par le compilateur :

- Ambiguïté de surcharge car le constructeur déjà existant possède le même prototype **(int, float)**

- Interdiction d'initialiser **id** dans la liste d'initialisation car ce n'est pas un attribut de cette classe dérivée (mais on a le droit de passer le paramètre **id** au constructeur de **Sensor**)

- il n'existe pas de constructeur par défaut par défaut pour la classe **Sensor** car il en existe déjà un avec un paramètre **int**. Or il est nécessaire pour ce nouveau constructeur qui n'appelle pas le constructeur existant de **Sensor**.

2.3) Revenons au code initial de la page précédente. On enlève le mot **`virtual`** de la ligne 9.
Question : Indiquer si la compilation de ce programme produit toujours un exécutable :
Si vous répondez « oui », préciser s'il y modification de l'affichage obtenu à l'exécution.
Si vous répondez « non », préciser la cause de l'erreur

[All] Oui, un exécutable est produit. Le mot clef **virtual** n'est pas obligatoire pour produire un exécutable MAIS le comportement du programme change car seul le destructeur de la classe **Sensor** est appelé pour l'exécution de la ligne 49 car il n'est plus **virtual**. L'appel de la ligne 49 produit seulement cet affichage :

[W]	`destroy Sensor : 2025`
[B]	`destroy Sensor : 2500`
[Y]	`destroy Sensor : 2250`
[S]	`destroy Sensor : 2052`

W=blanc, B=bleu, S=saumon, Y=jaune, All = même réponse pour tous

3) (9 pts) Héritage Multiple

Ce code **ex3.cc** compile sans warning avec -std=c++11 -Wall. Son exécution se termine normalement.

```
1   #include <iostream>
2   using namespace std;
3   class Base {
4   public :
5         Base() {cout << " Base constructor " <<  endl ;}
6         virtual ~ Base() {cout << " Base destructor " <<  endl ;}
7         virtual void display(){cout << " Base display " <<  endl ;}
8         void show(){cout << " Base show " << endl ;}
9   };
10
11  class Derived1 : public virtual Base {
12  public :
13        Derived1() {cout << " Derived1 constructor " <<  endl ;}
14        ~ Derived1() {cout << " Derived1 destructor " <<  endl ;}
15        void display()override {cout << " Derived1 display "
16                                     <<  endl ;}
17        void show(){cout << " Derived1 show " <<  endl ;}
18  };
19
20  class Derived2 : public virtual Base {
21  public :
22        Derived2() {cout << " Derived2 constructor " <<  endl ;}
23        ~ Derived2() {cout << " Derived2 destructor " <<  endl ;}
24        void display()override {cout << " Derived2 display "
25                                     <<  endl ;}
26        void show(){cout << " Derived2 show " <<  endl ;}
27  };
28
29  class Final : public Derived1 , public Derived2 {
30  public :
31        Final() {cout << " Final constructor " <<  endl ;}
32        ~ Final() {cout << " Final destructor " <<  endl ;}
33        void display()override {cout << " Final display " <<  endl ;}
34  };
35
36  int main()
37  {
38      {
39          cout << "Block 1:" <<  endl ;
40          Base* b = new Derived1();
41          b->display();
42          b->show();
43          delete b;
44      }
45
46      {
47          cout << "Block 2:" <<  endl ;
48          Final f;
49          Base& ref = f;
50          ref.display();
51          ref.show();
52       }
53       return 0;
54  }
```

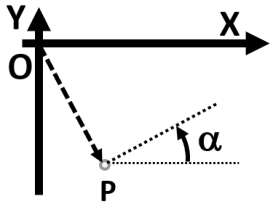
W=blanc, B=bleu, S=saumon, Y=jaune, All = même réponse pour tous

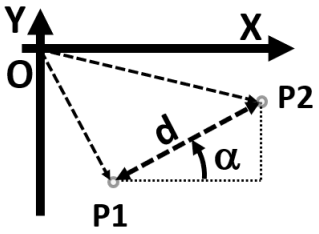
L'exécution de ce code produit plusieurs affichages dans le terminal. Utiliser cette table en indiquant dans la colonne de gauche le numéro de la ligne de code causant l'affichage et la colonne de droite pour montrer et JUSTIFIER le texte affiché.

N° ligne de code [All]	Texte affiché et *JUSTIFICATION* **[All]**
39	**`Block 1 :`** *Simple affichage d'une chaîne constante*
40 41 42 43	Allocation dynamique dans un pointeur **b** de la classe Base produisant d'abord le constructeur de la classe **Base** puis celui de la classe **Derived1** **`Base constructor`** **`Derived1 constructor`** Appel de la méthode **display** de **Derived1** qui car elle est **virtual** et est appelée via le pointeur **b** **`Derived1 display`** Appel de la méthode **show** de **Base** qui car elle n'est pas **virtual** **`Base show`** Appel du destructeur de **Derived1** qui car il est **virtual** et est appelée via le pointeur **b** ; ensuite le destructeur de **Base** est appelé en dernier. **`Derived1 destructor`** **`Base destructor`**
47	**`Block 2 :`** *Simple affichage d'une chaîne constante*
48 50 51 52	L'instance f de **Final** produit un appel unique au constructeur de **Base** puis aux constructeurs dont elle hérite dans l'ordre indiqué ligne 29, et enfin un appel au constructeur de **Final** est effectué : **`Base constructor`** **`Derived1 constructor`** **`Derived2 constructor`** **`Final constructor`** étant donné que la méthode **display** est **virtual**, et qu'elle est appelée *via une référence*, alors c'est la méthode de la classe **Final** qui est substituée et qui affiche : **`Final display`** Appel de la méthode **show** de **Base** qui car elle n'est pas **virtual** **`Base show`** Comme c'est la fin de l'instance **f** de la classe **Final**, et que le destructeur de **Base** est **virtual**, on appelle d'abord le destructeur de **Final** puis des destructeurs des 2 classes dérivées dont **Final** hérite *dans l'ordre inverse des constructeurs,* celui de **Derived2** avant celui de **Derived1** ; ensuite le destructeur de **Base** est appelé en dernier. **`Final destructor`** **`Derived2 destructor`** **`Derived1 destructor`** **`Base destructor`**

4) (8 pts) surcharge d'opérateur : soit le fichier **ex4.cc** dont le code partiel est listé ci-dessous. On y voit la définition d'une structure **S2d** pour placer un point (x,y) dans le plan 2D et d'une classe **MovingPoint** pour modéliser un point **p** du plan (ligne 16) qui se déplace selon la direction indiquée par l'angle **alpha** en rd (ligne 17 et illustration sur la droite) :

```
1   #include <cmath>
2
3   struct S2d{
4       S2d(double x=0., double y=0.): x(x),y(y){}
5       double x ;
6       double y ;
7   };
8
9   class MovingPoint {
10  public:
11        MovingPoint() = default;
12        MovingPoint(S2d p, double alpha=0)
13                    :p(p),alpha(alpha) {}
14        MovingPoint operator+(double d);
15  private:
16        S2d p;
17        double alpha;
18  };
19
20  // il manque la définition de la surcharge de l'opérateur +
21
22  int main(){
23        S2d p(2., -4);
24        MovingPoint p1(p,0.3);
25        MovingPoint p2;
26        p2 = p1 + 5. ;
27        return 0;
28  }
```





On demande d'écrire le code source de la surcharge de l'opérateur de l'addition dont le prototype est visible ligne 14. A titre d'exemple, cette surcharge est utilisée ligne 26 pour mettre à jour la valeur de l'instance **p2** obtenue en déplaçant **p1** dans la direction indiquée par **alpha** d'une distance **d** comme illustré avec la seconde figure.

```
MovingPoint MovingPoint::operator+(double d)
{
    // on prend la valeur absolue par sécurité
    // car une distance est toujours positive en Math :
    //     https://en.wikipedia.org/wiki/Metric_space
    d = fabs(d);

    S2d v(cos(alpha), sin(alpha));
    p.x += d*v.x;
    p.y += d*v.y;

     return *this;
}
```