TESTC++ Sem2 MT & EL 05.07.2021 EPFL / R. Boulic

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

1) (10 pts) Surcharge des opérateurs [All]

1.1) Les déclarations C++11 suivantes sont correctes.

```
1   class A
2   {
3   private:
4         int a;
5   public:
6         const A operator+(A const& a) const;
7   };
8
9   class B {
10  private:
11        int b;
12  };
13
14  class C {
15  private:
16        int c;
17  };
18
19  const A operator+(A const& a, A const& b);
20
21  const C operator+(C const& c, C const& b);
```

Choisir pour chaque classe la surcharge d'opérateur qu'elle implémente :

Classe	Entourer la réponse retenue **puis brièvement justifier votre réponse**			
A	Externe	Interne	Les deux	Aucune
	Ligne 6 : surcharge interne Ligne 19 : surcharge externe car opérandes de la classe A			
B	Externe	Interne	Les deux	Aucune
	Aucun opérateur surchargé ne travaille avec des opérandes de la classe B			
C	Externe	Interne	Les deux	Aucune
	Ligne 21 : les deux opérandes appartiennent à la classe C			

1.2) Dans ce code, les détails de la classe D ne sont pas fournis.

TESTC++ Sem2 MT & EL 05.07.2021 EPFL / R. Boulic
W=blanc, B=bleu, S=saumon, Y=jaune, All = même réponse pour tous

```
1   #include <iostream>
2
3   int main ()
4   {
5      D d1, d2;
6      int e(99);
7
8      d1 += d2;
9
10     std::cout << d1;
11
12     e = 3 + d1;
13
14     e = d1 + 3;
15
16     return 0;
17  }
```

Choisir quelles surcharges d'opérateur la classe D peut définir pour que le code puisse compiler. Si les deux types de surcharges peuvent être définis, il faut indiquer cette réponse.

ligne	Entourer la réponse retenue **puis brièvement justifier votre réponse**		
8	Externe	Interne	Les deux
	La surcharge externe peut utiliser la surcharge interne pour modifier l'opérande gauche		
10	Externe	Interne	Les deux
	Car on ne peut pas modifier la classe de l'opérande gauche pour ajouter une redéfinition à cette classe définie par le langage C++		
12	Externe	Interne	Les deux
	Il s'agit de l'addition d'un **int** (opérande gauche) à une instance de **D** (opérande droit). Or on ne peut pas ajouter une surcharge interne à l'opérateur + pour un type de base.		
14	Externe	Interne	Les deux
	Dans ce cas, une instance de **D** est l'opérande gauche de l'addition (donc on peut surcharger l'addition en interne) et l'entier est l'opérande de droite (donc on peut surcharger aussi en externe).		

1.3) Le code ci-dessous présente une erreur de compilation. Décrire l'erreur et expliquer comment la corriger en détaillant les modifications ou ajout de code (préciser le numéro de la ou des lignes modifiées ; indiquer où insérer d'éventuelles nouvelles lignes de code).
Remarque : il n'est pas autorisé de modifier la fonction main().

```
1   #include <iostream>
2
3   class G
4   {
5   public:
6      G(int g): g(g) {};
7      const int operator+(int val) const { return g + val; }
8   private:
9      int g;
10  };
11
12  std::ostream& operator<<(std::ostream& sortie, G const& g)
13  {
14     sortie << g.g;
15     return sortie;
16  }
17
18  int main()
19  {
20     G g1(3);
21
22     std::cout << "Somme: " << g1 + 2;
23     return 0;
24  }
```

L'erreur est à la ligne 14
Justification : l'attribut private **g** ne peut pas être accédé dans la portée de la redéfinition externe de l'opérateur << car une telle surcharge externe n'est PAS dans la portée de la classe **G**.

Corrections acceptées (au choix) : complément d'information

1) Ajouter une méthode **public** getter qui renvoie la valeur de l'attribut **g** et l'appeler dans la surcharge de << :
   Entre les lignes 6 et 7 : `int get_g() const { return g ; }`
   Remplacer la ligne 14 par : `sortie << g.get_g() ;`

2) Ajouter une méthode **public** print qui affiche l'attribut **g** et est appelée dans la surcharge de << :
   Entre les lignes 6 et 7 : `void print(std::ostream& sortie) const { sortie << g; }`
   Remplacer la ligne 14 par : `g.print(sortie);`

3) Associer la surcharge externe de << comme **friend** de la classe **G** en ajoutant sa déclaration précédée de **« friend »** entre les ligne 7 et 8 (note : cela ne fait pas de différence d'apparaître en public ou en private).

2) (6 pts) Erreur sémantique

Ce code compile sans warning en C++11 ; cependant il contient une erreur sémantique.

```
1   #include <iostream>
2   #include <vector>
3
4   typedef std::vector<int> MyList;
5
6   //cette fonction modifie v en multipliant chacun de
7   //ses éléments par la valeur s
8   void mult_s(MyList& v, double s);
9
10  int main()
11  {
12       MyList v={1,2,3};
13       double s(2.);
14
15       for(auto elem : v)
16          std::cout << elem << ' ';
17       std::cout << std::endl;
18
19       mult_s(v,s);
20
21       for(auto elem : v)
22          std::cout << elem << ' ';
23       std::cout << std::endl;
24
25       return 0;
26  }
27
28  void mult_s(MyList& v, double s)
29  {
30       for(auto elem : v)
31         elem *= s ;
32  }
```

Trouver l'erreur sémantique présente dans ce code en précisant les résultats d'affichage des lignes 15-17 et 21-23. Indiquer comment elle peut être corrigée. Affichage

[W]: 15-17 :**1 2 3** *suivi par un seul* ***endl***
21-23 :**1 2 3** *suivi par un seul* ***endl***

[B] : code différent pour les lignes 12-13 :

```
MyList v={2,4,6};
double s(0.5);
```

15-17 :**2 4 6** *suivi par un seul* ***endl***
21-23 :**2 4 6** *suivi par un seul* ***endl***

[Y] : code différent pour les lignes 12-13 :

```
MyList v={3,2,1};
double s(2.);
```

15-17 :**3 2 1** *suivi par un seul* ***endl***
21-23 :**3 2 1** *suivi par un seul* ***endl***

[S] : code différent pour les lignes 12-13 :

```
MyList v={6,4,2};
double s(0.5);
```

15-17 :**6 4 2** *suivi par un seul* ***endl***
21-23 :**6 4 2** *suivi par un seul* ***endl***

[All] nature de l'erreur sémantique :
le vector **v** n'est jamais modifié quelque soit la valeur de **s**, contrairement à la description de la fonction **mult_s** en ligne 6.

En effet, il n'est pas suffisant de passer le vector par référence (ce qui est fait). Le problème provient de la boucle à la ligne 30 car la variable **elem** définie dans cette boucle est une **variable locale** à la boucle qui ne modifie **PAS** l'élément de **v** qui sert à l'initialiser à chaque passage dans la boucle.

A la ligne 31, seule la variable **elem** est modifiée sans modifier l'élément du vector avec lequel elle a été initialisée.

[All] correction :

Indiquer que la boucle range for travaille par **référence** pour que **elem soit une référence sur un élément du vector** à chaque passage dans la boucle. Pour cela il faut ajouter **&** après le mot clef **auto** à la ligne 30.

Une correction avec une boucle for accédant explicitement à chaque élément du vector avec une variable d'indice **i** est aussi acceptée comme correction :

```
for(unsigned i(0) ; i < v.size() ; ++i)
    v[i] *= s ;
```

Complément :

le fait que la fonction **mult_s** multiplie un entier avec une valeur de type double n'est pas en soi une erreur sémantique. Rien dans le contexte ni dans les valeurs fournies pour l'exemple n'indique que l'éventuelle approximation entière du résultat de la multiplication par s soit un problème pour ce code.

D'ailleurs, aucun changement des types de v ou de s ne supprime la véritable erreur sémantique car v n'est jamais modifié par **mult_s**.

3) (9 pts) Polymorphisme : ce code compile sans warning en C++11.

```
1   #include <iostream>
2   using namespace std;
3
4   class A {
5   public:
6         virtual string foo() {
7               return "foo";
8         }
9         virtual string bar() {
10              return "bar";
11        }
12        void print() {
13              cout << foo() << ' ' << bar() << endl;
14        }
15  };
16
17  class B: public A{
18  public:
19        string foo() override {
20              return "boo";
21        }
22
23        void print() {
24              cout << "B ";
25              A::print();
26        }
27  };
28
29  class C: public B {
30  public:
31        string bar() override {
32              return "car";
33        }
34        void print() {
35              cout << "C ";
36              A::print();
37        }
38  };
39
40  int main() {
41        A a;
42        a.print();
43
44        B b;
45        b.print();
46
47        C c;
48        c.print();
49
50        B& br = c;
51        br.print();
52
53        return 0;
54  }
```

TESTC++ Sem2 MT & EL 05.07.2021 EPFL / R. Boulic

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

Indiquer l'affichage obtenu pour chacune des lignes de code suivantes ; [All]

une justification est exigée pour chaque cas

tous les affichages indiqués ci-dessous sont suivi d'un passage à la ligne

Ligne 42 :

```
foo bar
```

car on appelle la méthode **print** de A qui affiche les valeurs renvoyées par **foo()** qui renvoie **foo** et **bar()** qui renvoie **bar** . Ces deux méthodes appartiennent à la classe **A**

Ligne 45 :

```
B boo bar
```

Car B hérite de A et la méthode **print** est redéfinie dans la classe B (masquage de la méthode de A).

Cela explique l'affichage du B. Ensuite cette méthode **print** de B appelle explicitement la méthode print de A à l'aide de l'opérateur de redéfinition de portée. Celle-ci appelle deux méthodes virtuelles dont **foo()** est redéfinie (substituée) dans la classe B. l'affichage est donc **boo** provenant de la méthode virtuelle substituée et **bar** provenant de la méthode définie dans la classe **A**

Ligne 48 :

```
C boo car
```

Car C hérite de B et la méthode **print** est redéfinie dans la classe C (masquage de la méthode de B).

Cela explique l'affichage du C. Ensuite cette méthode **print** de C appelle explicitement la méthode print de A à l'aide de l'opérateur de redéfinition de portée. Celle-ci appelle deux méthodes virtuelles dont **foo()** est redéfinie (substituée) dans la classe B mais pas dans la classe C. l'affichage est donc **boo** provenant de la méthode virtuelle substituée dans B et **car** provenant de la méthode **bar()** redéfinie (substituée) dans la classe **C**

Ligne 51 :

```
B boo car
```

La variable **br** est une **référence** de la classe B initialisée avec une référence sur une variable de la classe C. Cela explique l'affichage du **B** car **br** est de la classe B et la méthode **print** est redéfinie dans la classe B (elle masque la méthode de même nom de A). Cependant la méthode **print** n'est PAS virtuelle, c'est pourquoi on n'appelle PAS la redéfinition (substitution) de **print** par la classe C.

Ensuite cette méthode **print** de B appelle explicitement la méthode **print** de A à l'aide de l'opérateur de redéfinition de portée. Celle-ci appelle deux méthodes virtuelles dont **foo()** est redéfinie (substituée) dans la classe B mais pas dans la classe C. l'affichage est donc **boo** provenant de la méthode virtuelle substituée dans B et **car** provenant de la méthode **bar()** redéfinie (substituée) dans la classe **C.** Ceci est possible car **br** est une **référence** sur une instance de C ce qui nous autorise à accéder à la méthode virtuelle de **bar()** redéfinie (substituée) par la classe C.

4) (10 pts) Héritage simple et multiple : ce code compile sans warning en C++11.

```
1   #include <iostream>
2   using namespace std;
3
4   class A
5   {
6       public:
7           A(int x=10, int y=20): x_(x), y_(y) {};
8           ~A() {cout << " A destroyed " << endl; }
9           void affiche() const
10            {cout << "x = " << x_ << ", y = " << y_ << endl;}
11
12      protected:
13          int x_;
14
15      private:
16          int y_;
17  };
18
19  class B : public A
20  {
21      public:
22          void set_x(int x){x_ = x;}
23          int get_x() const {return x_;}
24
25      protected:
26          int x_=30;
27  };
28
29  class C : public virtual A
30  {
31      public:
32          C(): pt_(nullptr) {};
33          ~C() {cout << " C destroyed " << endl; }
34          int* get_pt() const {return pt_;}
35          void set_pt(int z) {pt_ = &x_; *pt_ = z;}
36      protected :
37          int* pt_;
38  };
39
40  class D : public virtual A
41  {
42      public:
43          D(): pt_(nullptr) {};
44          ~D() {cout << " D destroyed " << endl; }
45          void set_pt(int& z) {pt_ = &z;}
46      protected :
47          int* pt_;
48  };
49
50  class E : public D, public C
51  {
52      public :
53          ~E() {cout << " E destroyed " << endl; }
54  };
```

```
55  int main()
56  {
57        { // ce bloc est destiné à limiter la portée de b
58              B b;
59              cout << b.get_x() << endl;     // Q0
60              b.set_x(2);
61              b.affiche();                   // Q1
62              cout << b.get_x() << endl;     // Q2
63        }
64
65        { // ce bloc est destiné à limiter la portée de c
66              C c;
67              //cout << c.x_ << endl;        // Q3
68              //cout << c.y_ << endl;        // Q3
69
70              int reponse = 42;
71              c.set_pt(reponse);
72              reponse = 9;
73             //int* p(c.get_pt());          // Q4
74              //cout << (p ? *p:0) << endl; // Q4
75        }
76
77        { // ce bloc est destiné à limiter la portée de e
78              //int reponse = 42;            // Q5
79              E e ;
80              //e.set_pt(reponse);           // Q5
81        } // Q6
82        return 0;
    }
```

Chaque question est liée à certaines lignes du code indiquées par un commentaire en fin de ligne(s).

Si les instructions d'une question sont commentées (ex : Q3, Q4 et Q5) vous devez indiquer si le code compile quand on enlève ces commentaires et si oui, il faut indiquer le résultat de l'exécution. Si les instructions ne compilent pas, ces lignes restent en commentaire.

Question / Ligne(s)	Affichage obtenu ou description d'erreur de compilation Tous les affichages sont suivis par un passage à la ligne
Q0, ligne 59	**[All]** **30**
Q1, ligne 61	**[All]** **x = 10 , y = 20**
Q2, ligne 62	Initialisations importantes pour les affichages suivants : [W] [B] [Y] [S] Ligne 60: **2** **9** **42** **42** Affichage : [W] [B] [Y] [S] **2** **9** **42** **42**

Q3, lignes 67-68	**[All]** **Erreur de compilation pour les 2 lignes** : Ligne 67 : l'attribut protected **x_** est accédé en dehors de la portée de la classe C Ligne 68 : l'attribut private **y_** ne peut jamais être accédé par une instance de C
Q4, ligne 73-74	Initialisations importantes pour les affichages suivants : [W] [B] [Y] [S] Ligne 70: **42** **2** **9** **2** Ligne 72: **9** **42** **2** **9** **[W] [B] [Y] [S]** **42** **2** **9** **2**
Q5, lignes 78-80	**[All]** **Erreur de compilation pour l'appel à la ligne 80** L'appel est ambigu car la méthode est redéfinie par les deux classes parentes D et C de l'instance de la classe E
Q6, ligne 81 **Quel affichage est produit en quittant ce bloc ?**	**[All]** **E destroyed** **C destroyed** **D destroyed** **A destroyed** Complément : les destructeurs sont appelés dans l'ordre inverse des constructeurs de la classe E.