TESTC++PoP Sem2 MT & EL 06.05.2022 EPFL / R. Boulic

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

1) (6 pts) Analyse de code

```
1   #include <iostream>
2
3   class Vector3D{
4   public:
5        Vector3D(const Vector3D &v){
6           x = v.x; y = v.y; z = v.z;
7        }
8        int getX() { return x; }
9        int getY() { return y; }
10       int getZ() { return z; }
11  private:
12       int x, y, z;
13  };
14
15  int main(){
16       Vector3D v1;
17       Vector3D v2(v1);
18       std::cout << "x = "   << v2.getX()
19                 << ", y = " << v2.getY()
20                 << ", z = " << v2.getZ();
21       return 0;
22  }
```

1.1) Choisir une réponse parmi les suivantes All

Réponse	Description	Cocher une case
A	On obtient un exécutable qui affiche : `x = 0, y = 0, z = 0`	
B	On obtient un exécutable qui affiche des valeurs quelconques pour les attributs x, y et z	
C	L'exécution se termine anormalement, par exemple avec un message `segmentation fault`	
D	Une erreur est détectée à l'étape de la **compilation**	X
E	Un erreur est détectée à l'étape de **l'édition de liens**	

1.2) **Justifier votre réponse** à la question précédente ; si vous avez choisi les réponses C à E alors précisez la nature de l'erreur et les instructions qu'il faut modifier ou ajouter pour supprimer le problème. All

Le compilateur détecte l'erreur de compilation à la ligne 16 qui declare v1 car il n'existe pas de constructeur par défaut par défaut. Cela est dû à l'existence d'un autre constructeur.

On corrige ce problème en ajoutant un constructeur par défaut, après la ligne 4 :

```
Vector3D() { x=0 ; y=0 ; z=0 ; }
```

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

2) (9 pts) Surcharge d'opérateurs

Ce code compile sans warning en C++11 et s'exécute normalement

```
1   #include <iostream>
2   using namespace std;
3
4   class Complex
5   {
6   public:
7       Complex(): m_real(0.0), m_imag(0.0){ }
8       Complex(double real, double imag): m_real(real), m_imag(imag){ }
9       Complex(double real): m_real(real), m_imag(0.0){ }
10
11      friend Complex operator+(const Complex &c1, const Complex &c2);
12      friend ostream& operator<<(ostream& os,const Complex& c);
13
14  private:
15      double m_real;
16      double m_imag;
17  };
18
19  Complex operator+(const Complex &c1, const Complex &c2)
20  {
21      Complex c;
22      c.m_real = c1.m_real + c2.m_real;
23      c.m_imag = c1.m_imag + c2.m_imag;
24      return c;
25  }
26
27  ostream& operator<<(ostream& os, const Complex& c)
28  {
29      return os << c.m_real << " + " << c.m_imag << 'i' << endl;
30  }
31
32  int main()
33  {
34      Complex c1(1., 2.);
35      Complex c2(c1 + 3.4);
36      Complex c3(3.4 + c2);
37
38      cout << c1;
39      cout << c2;
40      cout << c3;
41
42      return 0;
43  }
```

2.1) indiquez le résultat de l'exécution de chacune des lignes de code suivantes et justifier brièvement comment la valeur de la variable affichée est initialisée :

Ligne 38 : [All] `1 + 2i`

Justifier en détaillant l'initialisation de c1 à la ligne 34: [All]

le constructeur de la ligne 8 affecte 1 à l'attribut m_real et 2 à l'attribut m_imag

Ligne 39 : [W] `4.4 + 2i`

Justifier en détaillant l'initialisation de c2 à la ligne 35: [All]

L'unique surcharge (externe) de l'opérateur + (lignes 19-25) demande 2 opérandes Complex ; l'opérande droit est donc initialisé avec le constructeur de la ligne 9 qui initialise un Complex ayant

m_real à 3.4 et m_imag à 0. Les attributs de c1 et de ce nouveau Complex sont ensuite additionnés pour obtenir c2 avec m_real valant 4.4 = 3.4+1 et m_imag valant 2 = 2+0.

Ligne 40 : [W] `7.8 + 2i`
Justifier en détaillant l'initialisation de c3 à la ligne 36 : [All]
L'unique surcharge (externe) de l'opérateur + (lignes 19-25) demande 2 opérandes Complex ; l'opérande gauche est donc initialisé avec le constructeur de la ligne 9 qui initialise un Complex ayant m_real à 3.4 et m_imag à 0. Les attributs du nouveau Complex sont ensuite additionnés à c2 pour obtenir c3 avec m_real valant 7.8 = 3.4+4.4 et m_imag valant 2 = 0+2.

[B]
Ligne 39 : `3.4 + 2i`
Ligne 40 : `5.8 + 2i`

[S]
Ligne 39 : `2.4 + 2i`
Ligne 40 : `3.8 + 2i`

[Y]
Ligne 39 : `5.3 + 2i`
Ligne 40 : `9.6 + 2i`

2.2) Le mot-clef **friend** des lignes 11 et 12 est-il nécessaire ? Que se passe-t-il si on le supprime ?[All]
Oui car la surcharge est externe ; la fonction ne peut donc pas accéder aux attributs de la classe Complex s'il n'y a pas le mot clef `friend`. Cela produit une erreur de compilation.

La suite de l'exercice cherche à <u>modifier le code pour ne pas utiliser le mot clef friend.</u>
Le code de la fonction main() ne doit pas changer.

2.3) La surcharge de l'opérateur << peut-elle être effectuée avec une surcharge interne ? Selon votre réponse indiquez ci-dessous le code qu'il faut supprimer/écrire pour pouvoir surcharger l'opérateur << sans utiliser le mot clef **friend** et obtenir le même résultat d'exécution que pour la question 2.1. [All]
Seulement si nous sommes les concepteurs de la classe de l'opérande gauche, ce qui n'est pas le cas. On opte donc pour une surcharge externe.
Il faut enlever la ligne 12 et, par exemple, ajouter deux getters à partir de la ligne 10 qui sont appelés dans la fonction de la surcharge externe à la ligne 29:

```
double get_r() const {return m_real ;}
double get_i() const {return m_imag ;}
```

ligne 29 : `return os << c.get_r() << " + " << c.get_i() << 'i' << endl;`

2.4) La surcharge de l'opérateur + peut-elle être effectuée avec une surcharge interne dans ce programme ? Selon votre réponse indiquez ci-dessous le code qu'il faut supprimer/écrire pour pouvoir surcharger l'opérateur + sans utiliser le mot clef **friend** et obtenir le même résultat d'exécution que pour la question 2.1[All]
Non à cause de la ligne 36 dont l'opérande gauche n'est pas un Complex. Il faut continuer à utiliser une surcharge externe ayant le même prototype.
Il faut enlever les lignes 11, 22 et 23, et utiliser les deux getters définis plus haut et le constructeur de la ligne 9 pour remplacer la ligne 21 par :

```
Complex c(c1.get_r()+c2.get_r(), c1.get_i()+c2.get_i()) ;
```

3) (6 pts) Correction de code :

Ce code compile sans warning en C++11 mais produit une erreur à l'exécution.

```
1   #include <iostream>
2
3   class A
4   {
5   public:
6       A(): val(0), p(nullptr) {}
7       A(int v): val(v), p(new int(val)) {}
8       ~A() { delete p; }
9       void print () { if(p) std::cout << *p << std::endl; }
10  private:
11      int val;
12      int *p;
13  };
14
15  int main ()
16  {
17      A a1(10);
18      {
19          A a2;
20          a2 = a1;
21      }
22      a1.print();
23
24      return 0;
25  }
```

3.1) Pour quelle raison une erreur est-elle produite à l'exécution ?

Remarque : on ne demande pas le message d'erreur affiché par le système au moment de l'exécution mais seulement ce qui va causer un comportement incorrect à l'exécution. [All]

Il y a d'abord une allocation dynamique de un seul **int** initialisé avec la valeur **val** au moment de la déclatation de **a1**. Puis l'affectation de la ligne 20 effectue une copie superficielle de **a1** vers **a2**, ce qui copie l'adresse du pointeur **p** de **a1** vers celui de **a2**. A la sortie du bloc (ligne21) a2 n'existe plus et le destructeur (ligne 8) est appelé, ce qui va libérer le bloc mémoire pointé par le pointeur **p** de **a2** mais aussi par le pointeur **p** de **a1**. C'est la cause initiale des problèmes ultérieurs à l'exécution.

Supplément non demandé : plus tard **a1** va continuer à utiliser la même adresse mémoire alors que le bloc est déjà libéré (ligne 22) ce qui ne devrait pas être fait. Un autre problème est le second appel de **delete** sur le même bloc mémoire déjà libéré quand le destructeur est appelé sur **a1** à la fin de main() ; c'est ce qui cause le message d'erreur du système.

3.2) Quelle méthode faut-il ajouter à la classe A pour que l'exécution ne produise plus de message d'erreur ? **On ne peut pas modifier la fonction main()** . Fournir sa définition complète ci-dessous : [All] Puisqu'on ne peut pas changer main() il faut écrire une surcharge de l'opérateur d'affectation pour effectuer une copie profonde à la ligne 20 (et on devrait aussi écrire un constructeur de copie qui effectue le même travail). Cette définition peut être ajoutée après la définition de la méthode **print** dans la déclaration de la classe. On peut aussi ne mettre que la déclaration de la surcharge et externaliser sa définition.

```
void operator=(const A& other) { // Le type renvoyé peut aussi être A&
    val = other.val;
    p   = new int(val);
}
```

Cette version est suffisante pour ce programme mais pas dans le cas général où il produit une fuite de mémoire quand l'opérande gauche possède une valeur de **p** différente de **nullptr**. **p** doit donc être testé et si c'est le cas, il faut remplacer la seconde instruction par : `*p = *(other.p);`

4) (8 pts) Polymorphisme :

ce code compile sans warning avec l'option -std=c++11

```
1   #include <iostream>
2   using namespace std;
3
4   class BasicRobot {
5   public:
6        BasicRobot()  { f(); }
7        ~BasicRobot() { f(); }
8        void reboot() { f(); }
9   protected:
10       virtual void f() { cout << "Bip" << endl; }
11  };
12
13  class MobileRobot : public BasicRobot {
14  public:
15       MobileRobot() { f(); }
16       ~MobileRobot() { f(); }
17  protected:
18       virtual void f() { cout << "Vroum" << endl; }
19  };
20
21  class EmergencyRobot : public MobileRobot {
22  protected:
23       void f() { cout << "Pin-pon" << endl; }
24  };
25
26  int main(void) {
27       BasicRobot* pRobot = new EmergencyRobot;          //Q0
28       //pRobot->reboot();                               //Q1
29       //delete pRobot;                                  //Q2
30       return 0;
31  }
```

Chaque question est liée à certaines lignes du code indiquées par un commentaire en fin de ligne. Si l'instruction d'une question est commentée (Q1 et Q2) vous devez indiquer si le code s'exécute sans erreur quand on enlève le commentaire et, si oui, il faut indiquer le résultat de l'exécution. Si l'instruction ne s'exécute pas normalement la ligne reste en commentaire. [All]

Question / Ligne(s)	Affichage obtenu[1] ou description de la cause de l'erreur
Q0, ligne 27	**Bip** **Vroum**
Q1, ligne 28	Il n'y a pas d'erreur car **reboot** est une méthode de la classe **BasicRobot**. La méthode **reboot** appelle la méthode virtuelle **f()** qui, grâce au pointeur sur l'instance de la classe **EmergencyRobot**, appelle la méthode f() redéfinie à la ligne 23, d'où l'affichage de : **Pin-pon**
Q2, ligne 29	Il n'y a pas d'erreur car **pRobot** a été alloué dynamiquement. Le destructeur de **BasicRobot** n'étant pas virtuel il y a seulement l'appel du destructeur de cette classe qui appelle sa méthode **f**, d'où l'affichage de :

[1] Le texte surligné en bleu n'était pas exigé ; c'est pour comprendre le résultat de l'exécution

	Bip[2]

5) (6 pts) Héritage multiple :

5.1) Ce code produit une erreur de compilation.

```
1   #include <iostream>
2
3   class X {
4   public:
5         X() { b = 2; }
6         int b;
7   };
8
9   class Y : public X {
10  public:
11        Y() { b += 3; }
12  };
13
14  class Z : public X {
15  public:
16        Z() { b *= 4; }
17  };
18
19  class M : public Y, public Z {};
20
21  int main() {
22        M* m = new M();
23        std::cout << m->b << std::endl ;
24        return 0;
25  }
```

Préciser la nature de l'erreur de compilation détectée dans ce programme : [All]
Ligne 23, l
m->b est ambigu car le compilateur ne peut pas savoir de quelle copie de l'attribut **b** il s'agit car l'instance pointée par **m** obtient cet attribut deux fois.

5.2) Corriger ce problème en conservant la hiérarchie de classes et sans modifier main().

Pour avoir un seul attribut **b** pour l'instance pointée par **m**, il faut mettre en place l'héritage virtuel en ajoutant le mot clef **virtual** aux lignes 9 et 14 :

```
class Y : public virtual X {
class Z : public virtual X {
```

5.3) Quel est l'affichage obtenu après la correction apportée par la question précédente ?
Justifier comment cet affichage est obtenu.
[All] La construction de l'instance pointée par m est effectuée dans l'ordre des classes indiquées à la ligne 19 : d'abord le constructeur de la superclasse **X** puis **Y** puis **Z.**

[W] : b vaut 2 auquel on ajoute 3 d'où la valeur de 5 qui est multipliée par 4 qui donne 20 (affichée)
[B] : b vaut 2 auquel on ajoute 4 d'où la valeur de 6 qui est multipliée par 3 qui donne 18 (affichée)

[2] Les points ne sont pas donnés s'il y a du texte supplémentaire envoyé sur **cout**

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

[S] : b vaut 2 auquel on ajoute 3 d'où la valeur de 5 qui est multipliée par 5 qui donne 25 (affichée)

[Y] : b vaut 2 auquel on ajoute 5 d'où la valeur de 7 qui est multipliée par 3 qui donne 21 (affichée)