TESTC++ Sem2 MT & EL 05.05.2023 EPFL / R. Boulic
W=blanc, B=bleu, S=saumon, Y=jaune, All = même réponse pour tous

1) (7 pts) Classe : soit le fichier **ex1.cc** dont le code est listé ci-dessous

```
1   #include <iostream>
2
3   using namespace std;
4
5   class A
6   {
7       static int a;
8   public:
9       void show()
10      {
11          a++;
12          cout<<"a: "<<a<<endl;
13      }
14  };
15
16  int A::a = 5;
17
18  int main(int argc, char* argv[])
19  {
20       A a;
21       return 0;
22  }
```

1.1) On compile le fichier ex1.cc avec la commande : **`g++ -std=c++11 ex1.cc -o ex1`**

Choisir une réponse parmi les suivantes [All]

Réponse	Description	**Cocher une seule case**
A	L'exécution affiche : **`a: 5`**	
B	L'exécution affiche : **`a: 6`**	
C	L'exécution se termine normalement sans rien afficher	X
D	L'exécution se termine anormalement, par exemple avec un message **`segmentation fault`**	
E	Une erreur est détectée à l'étape de la **compilation**	

1.2) **Justifier votre réponse**[1] à la question précédente ; si vous avez choisi les réponses D ou E alors précisez la nature de l'erreur et les instructions qu'il faut modifier ou ajouter pour supprimer le problème. [All]

L'instance **a** de la classe **A** déclarée ligne **20** fait appel au constructeur par défaut par défaut, ce qui ne pose aucun problème. La méthode **show**() peut tout à fait accéder et modifier **l'attribut de classe** qui s'appelle aussi **a** et qui est initialisée à **5**. Cependant cette méthode n'est jamais appelée. L'instruction de déclaration n'effectue aucune action particulière et on quitte la fonction **main()** dès l'instruction suivante. On a donc une exécution normale sans affichage.

[1] L'absence de justification limite la note maximum de cet exercice à 1 point

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

2) (7 pts) Correction de code :

Ce code produit une erreur à la compilation en C++11.

```
1   #include <iostream>
2   using namespace std;
3
4   class A {
5   public:
6         A();
7         A( int );
8         int getVal() { return val; }
9   protected:
10        int val;
11  };
12
13  A::A() {
14        cout << "Call A::A() constructor." << endl;
15        val = 0;
16  }
17
18  A::A( int i ) {
19        cout << "Call A::A( int ) constructor." << endl;
20        val = i;
21  }
22
23  class B {
24        A a;
25  public:
26       B():a(1){}
27        void show() {cout << val << endl ; }
28  };
29
30  int main() {
31      B b;
32      b.show();
33      return 0;
34  }
```

2.1) Quelle est la nature de l'erreur produisant l'erreur à la compilation ? [All]

L'instance **b** de la classe **B** « *possède un attribut* ***a*** » mais « *n'est pas dérivé* » de la classe **A**.

L'instruction de la ligne **27** n'est donc pas dans la portée de la classe **A**, ce qui produit l'erreur de compilation quand la méthode **show** essaie d'accéder à l'attribut **val** de la classe **A**.

2.2) Corriger le code de manière à pouvoir exécuter la fonction main() (**on ne peut pas modifier la fonction main())** . Indiquer ce que vous ajoutez/supprimez ; préciser où vous faites les modifications.

Il suffit d'utiliser la méthode publique **getVal** de la classe **A** sur l'attribut **a** de la classe **B**.

Ligne 27 :

```
cout << a.getVal() << endl ;
```

2.3) Montrer ce qui est affiché avec l'exécution de main avec le code corrigé.

White	`Call A ::A(int) constructor.` `1`
Blue	`Call A ::A(int) constructor.` `-1`
Yellow	`Call A ::A(int) constructor.` `0`
Salmon	`Call A ::A(int) constructor.` `2`

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

3) (7 pts) surcharge des opérateurs pour faciliter les opérations de cryptographie [All]

L'arithmétique modulaire sur des entiers non-signés est un élément central pour certaines opérations de cryptographie. L'addition et la multiplication sont effectuées comme avec les entiers mais, en plus, on applique l'opérateur modulo (%) au résultat.

Exemple : l'addition **5+7** donne **12** en arithmétique standard tandis qu'en arithmétique modulaire modulo **11** on obtient **1** comme résultat car **12%11** vaut **1**.

Cet exercice propose une classe **Zn** pour mémoriser une valeur **val** dans l'arithmétique modulaire modulo **mod**. Le code ci-dessous surcharge les opérateurs **+** et ***** pour effectuer ces opérations en arithmétique modulaire modulo **mod**.

Ce code compile sans warning avec les options -std=c++11 -Wall

```
1   #include <iostream>
2   using namespace std;
3
4   class Zn
5   {
6   public:
7       Zn(unsigned val, unsigned mod):val{val%mod},mod{mod}{}
8   private:
9       unsigned int val;
10      unsigned int mod;
11      friend ostream& operator<<(ostream& os, const Zn& zn);
12      friend Zn operator+(const Zn& lhs,const Zn& rhs);
13      friend Zn operator*(const Zn& lhs,const Zn& rhs);
14  };
15
16  Zn operator+(const Zn& lhs,const Zn& rhs)
17  {
18      return Zn((lhs.val+rhs.val) % lhs.mod,lhs.mod);
19  }
20  Zn operator*(const Zn& lhs,const Zn& rhs)
21  {
22      return Zn((lhs.val*rhs.val) % lhs.mod,lhs.mod);
23  }
24  ostream& operator<<(ostream& os, const Zn& zn)
25  {
26      os << "Zn("<<zn.val<<" mod "<<zn.mod<<")";
27      return os;
28  }
29
30  int main()
31  {
32        Zn a(3,7);
33        Zn b(6,11);
34        Zn c(5,13);
35        cout << "a="<<a<<" b="<<b <<" c="<<c<< endl; // Q1
36        cout << a+b+c << endl;                       // Q2
37        cout << a*b*c << endl;                       // Q3
38        cout << a+b*c << endl;                       // Q4
39  }
```

Rappel : les règles de priorité entre opérateurs s'appliquent aussi aux opérateurs surchargés.

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

3.1) Quel est l'affichage réalisé à la ligne avec le commentaire //Q1

a=Zn(3 mod 7) b=Zn(6 mod 11) c=Zn(5 mod 13)

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

3.2) Justifiez comment l'affichage de la question 3.1 est obtenu en détaillant la suite des appels

- Les constructeurs initialisent les attributs **val** et **mod**
- L'attribut **val** prend le résultat de l'expression **val%mod**
- L'opérateur **<<** est surchargé (externe) ; il peut accéder aux attributs car il est déclaré comme **friend** de la classe ligne 11. Il affiche **Zn(** *val* **mod** *mod***)** où *val* et *mod* sont les valeurs des attributs correspondants.

3.3) Quel est l'affichage réalisé à la ligne avec le commentaire //Q2

Zn(*0* **mod** *7***)**

3.4) Justifiez comment l'affichage de la question 3.3 est obtenu est détaillant les appels et les valeurs intermédiaires des calculs.

Les opérateurs externes << et + accèdent aux attributs car ils sont déclarés comme **friend** de la classe (lignes 11 et 12).
Associativité gauche-droite : d'abord a+b puis addition du résultat à c
a+b => (3,7) + (6,11) => ((3+6)%7 , 7) => (2,7)
résultat + c => (2,7) + (5,13) => ((2+5)%7 , 7) => (0, 7)

3.5) Quel est l'affichage réalisé à la ligne avec le commentaire //Q3

Zn(*6* **mod** *7***)**

3.6) Justifiez comment l'affichage de la question 3.5 est obtenu est détaillant les appels et les valeurs intermédiaires des calculs.

Les opérateurs externes << et * accèdent aux attributs car ils sont déclarés comme **friend** de la classe (ligne 11 et 13).
Associativité gauche-droite : d'abord a*b puis multiplication du résultat à c
a*b => (3,7) * (6,11) => ((3*6)%7 , 7) => (4,7)
résultat * c => (4,7) * (5,13) => ((4*5)%7 , 7) => (6, 7)

3.7) Quel est l'affichage réalisé à la ligne avec le commentaire //Q4

Zn(*4* **mod** *7***)**

3.8) Justifiez comment l'affichage de la question 3.7 est obtenu est détaillant les appels et les valeurs intermédiaires des calculs.

Tous les opérateurs externes accèdent aux attributs car ils sont déclarés comme **friend** de la classe (lignes 11, 12 et 13).
La multiplication est prioritaire par rapport à l'addition : d'abord b*c puis addition de a au résultat
b*c => (6,11) * (5,13) => ((6*5)%11 ,11) => (8,11)
a + résultat => (3,7) + (8,11) => ((3+8)%7 , 7) => (4, 7)

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

4) (7 pts) Héritage : soit le fichier **ex4.cc** dont le code est listé ci-dessous **[All]**

```
1   #include <iostream>
2
3   class Vehicle
4   {
5   public:
6       Vehicle(int wheels) : wheels_(wheels) {}
7
8       int getWheels() const
9       {
10          return wheels_;
11      }
12
13  private:
14      int wheels_;
15  };
16
17  class Car : public Vehicle
18  {
19  public:
20      Car() : Vehicle(4) {}
21
22      void honk()
23      {
24          std::cout << "Beep! Beep!" << std::endl;
25      }
26  };
27
28  int main()
29  {
30      Car myCar;
31      std::cout << "My car has " << myCar.getWheels()
32                                 << " wheels." << std::endl;
33      myCar.honk();
34
35      Vehicle myVehicle;
36      std::cout << "My vehicle has " << myVehicle.getWheels()
37                                     << " wheels." << std::endl;
38
39      return 0;
40  }
```

4.1) On compile le fichier ex4.cc avec la commande : **`g++ -std=c++11 ex4.cc -o ex4`**
Cette commande produit une erreur de compilation.

Quelle ligne de code / quelle instruction produit cette erreur ? Justifier votre réponse

Ligne 35 : déclaration de **myVehicle** ; il faudrait un constructeur par défaut mais il n'y en a pas. De plus cette classe ne peut pas avoir de **constructeur par défaut par défaut** puisqu'il existe déjà un constructeur qui demande un paramètre de type **int**.

4.2) Corriger la cause de l'erreur de compilation (**sans modifier le code de main()**). Vous pouvez au choix, soit modifier une ligne (*indiquer le numéro de cette ligne et recopiez ci-dessous la version correcte de la ligne*), soit ajouter du code supplémentaire (*indiquer où ce code doit être inséré et écrire ce code ci-dessous*).

Ligne **6** : il suffit de donner une valeur par défaut au paramètre **wheels** de ce constructeur

```
    Vehicle(int wheels=0) : wheels_(wheels){}
```

4.3) Quel est l'affichage qui doit être produit par l'exécution après correction de l'erreur de compilation ?

Avec le choix de valeur par défaut effectué dans la question précédente, on obtient :

```
My car has 4 wheels.
Beep ! Beep !
My vehicle has 0 wheels.
```

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

5) (7 pts) Héritage multiple : soit le fichier **ex5.cc** dont le code est listé ci-dessous

```
1   #include <iostream>
2
3   class Operable {
4   public:
5       Operable() : id(0) {}
6       virtual std::string getName() const = 0;
7       int getId() const {return id;}
8   protected:
9       int id;
10  };
11
12  class Movable : public Operable {
13  public:
14      virtual void move() const = 0;
15  };
16
17  class Drawable : public Operable {
18  public:
19      virtual void draw() const = 0;
20  };
21
22  class Shape : public Movable, public Drawable {
23  public:
24      Shape(std::string name) : name_(name) {}
25      void move() const override {
26          std::cout << getName() << " is moving."<< std::endl;
27      }
28      void draw() const override {
29          std::cout << getName() << " is being drawn."<< std::endl;
30      }
31      std::string getName() const override {
32          return name_;
33      }
34  protected:
35      std::string name_;
36  };
37
38  int main() {
39      Shape *myShape = new Shape("Circle");
40      Movable *movablePtr = myShape;
41      Drawable *drawablePtr = myShape;
42
43      std::cout << myShape->getName() << std::endl;
44      std::cout << myShape->getId()   << std::endl;
45      movablePtr->move();
46      drawablePtr->draw();
47
48      delete myShape;
49      return 0;
50  }
```

5.1) On compile le fichier ex5.cc avec la commande : **`g++ -std=c++11 ex5.cc -o ex5`**
Cette commande produit une erreur de compilation.

Quelle ligne de code / quelle instruction produit cette erreur ? Justifier votre réponse
(répondre en haut de la page suivante) **[All]**

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

Ligne **44** : l'appel à la méthode **getId()** est considéré comme ambigu par le compilateur car cette méthode pourrait avoir été rédéfinie dans l'une des classes parentes **Movable** et/ou **Drawable** et elle n'est pas redéfinie dans la classe **Shape,** contrairement aux 3 autres méthodes.

5.2) Corriger la cause de l'erreur de compilation (**sans modifier le code de main()**) ; préciser les numéros de ligne et écrire ci-dessous les lignes corrigées entières (*max deux lignes de code*) **[All]**

Solution incorrecte : il ne suffit pas de préciser explicitement qu'on appelle la méthode de la classe Operable avec la syntaxe suivante ligne 44 **:**

```
Std ::cout << myShape->Operable ::getId() << std ::endl ;
```

Solution véritable : en rendant la classe Operable virtual avec le mot clef virtual dans les déclarations des classes Drawable et Movable lignes 12 et 17 :

```
class Movable  : public virtual Operable {
class Drawable : public virtual Operable {
```

5.3) Quel est l'affichage qui doit être produit par l'exécution après correction de l'erreur de compilation ?

white	Blue	Yellow	Salmon

`Circle`
`0`
`Circle is moving`
`Circle is being drawn`
	`Circle`
	`1`
	`Circle is moving`
	`Circle is being drawn`
		`Circle`
		`-1`
		`Circle is moving`
		`Circle is being drawn`
			`Square`
			`0`
			`Square is moving`
			`Square is being drawn`

5.4) A quoi sert le mot clef **`override`** visible sur les lignes 25, 28 et 31 ? **[All]**

Qu'on redéfinit une méthode virtuelle héritée d'une superclasse. Grâce à l'indication **override** le compilateur vérifie s'il effectue bien une substitution de méthode virtuelle. Si aucun prototype n'existait dans la classe parente alors il signalerait une erreur.

5.5) Le fichier corrigé compile-t-il si on supprime le mot clef **`override`** sur les lignes 25, 28 et 31 ?

[All] Oui, ce mot clef est conseillé mais pas obligatoire.

5.6) Avec cette version corrigée du code, on recompile le code mais cette fois on demande l'affichage des warning avec : **`g++ -std=c++11  ex5.cc -Wall -o ex5`**

Un warning est affiché pour la ligne 48. Pour quelle raison ce warning est-il affiché ? **[All]**

Du fait de la ligne **6** où on voit une méthode **virtual**, le compilateur sait que le *polymorphisme* est possible dans cette hiérarchie de classes (indépendamment de l'héritage multiple). Dans ce cadre général, il est fortement recommandé de déclarer également comme **virtual** le destructeur de la superclasse **Operable**. Si cela n'est pas fait, l'action de **delete** sur un pointeur **Operable*** qui serait initialisé avec un pointeur **Shape*** ne produirait que l'appel du destructeur de **Operable**, ce qui pourrait produire une fuite de mémoire si les classes dérivées faisaient des allocations dynamiques supplémentaires. Cela dit *ça n'est pas le cas ici* car **delete** est appelé sur un pointeur **Shape***.

Par ailleurs le compilateur ne produit *aucun warning* sur le fait que les pointeurs **movablePtr** et **drawablePtr** ne sont pas ré-initialisés avec **nullptr** *après* la ligne 48[2].

5.7) **Sans modifier le code de main(),** que faut-il ajouter de plus pour faire disparaitre le warning ? indiquer où cet ajout est effectué. **[All]**

Ajouter un destructeur **virtual** à la superclasse **Operable**, par exemple entre les lignes 5 et 6 :

```
virtual ~Operable(){}
```

[2] Ceci n'est pas la réponse attendue ; cette information n'est pas demandée ; elle est fournie en complément