TESTC++ Sem2 MT & EL 17.05.24 EPFL / R. Boulic
W=blanc, B=bleu, S=saumon, Y=jaune, All = même réponse pour tous

1) (6 pts) surcharge des opérateurs : soit le fichier **ex1.cc** dont le code est listé ci-dessous

```
1  #include <iostream>
2  #include <string>
3
4  class Myclass {
5  public:
6    Myclass(int xx, const std::string &ss): x(xx), s(ss) {c++;}
7    static int get_c() {return c;}
8  private:
9    static int c;
10   int x;
11   std::string s;
12 };
13
14 int Myclass::c = 0;
15
16 Myclass operator+(const Myclass &obj1, const Myclass &obj2) {
17
18   return Myclass(obj1.x + obj2.x, obj1.s + obj2.s);
19 }
20
21 std::ostream &operator<<(std::ostream &stream, const Myclass &obj) {
22
23   return stream << "x: " << obj.x << " s: " << obj.s;
24 }
25
26 int main() {
27
28   Myclass ob1(10, "10"), ob2(20, "20"), ob3(30, "30"), ob4(40, "40");
29
30   std::cout << ob1 + ob3 + ob4 + ob2 << std::endl;
31
32   std::cout << Myclass::get_c() << std::endl;
33
34   return 0;
35 }
```

1.1) On compile le fichier ex1.cc avec la commande : **`g++ -std=c++11 ex1.cc -o ex1`**
Cette commande détecte des erreurs de compilation dès la ligne 18.
Quelle est la cause de ces erreur ?

[All] la première erreur de compilation est causée par la surcharge **externe** de l'opérateur+ car cette fonction n'a pas accès aux attributs **private** x et s

Le même problème existe pour l'autre surcharge externe, celle de l'opérateur <<

1.2) Il existe plusieurs manières de corriger ces erreurs SANS MODIFIER la fonction main() et tout en respectant le but des surcharges d'opérateurs. On demande de décrire, sur la page suivante, deux approches indépendantes qui suppriment ces erreurs (note : il en existe plus de deux).
Pour cela, indiquez la ou les lignes de code que vous modifiez ou que vous ajoutez. En cas d'ajout, indiquer où vous ajoutez du code « entre les lignes xx et yy » du code visible en page 2.
[All]

1.2.1.) Correction 1 : Approche: ajouter **deux méthodes getters** pour que les surcharges externes de + et << s'en servent pour accéder aux valeurs des attributs **x** et **s**.

Ajouter entre les lignes 6 et 7 ou ailleurs dans la partie publique de Myclass:

```
int          get_x() {return x ;}
std ::string get_s() {return s ;}
```

modifier la ligne 18 comme ceci :

```
return Myclass(obj1.get_x()+obj2.get_x(),obj1.get_s()+obj2.get_s());
```

modifier la ligne 23 comme ceci :

```
return stream << "x: " << obj.get_x() << " s: " << obj.get_s();
```

1.2.2.) Correction 2 : Approche: rendre les deux surcharges **friend** de la classe Myclass

Ajouter entre les lignes 6 et 7 ou ailleurs dans la partie publique de Myclass:

```
friend Myclass operator+(const Myclass &obj1, const Myclass &obj2);
friend std::ostream &operator<<(std::ostream &stream, const Myclass &obj);
```

Autres approches acceptées :

1) Ajouter une surcharge interne de + qui est appelée par la surcharge externe de +
   Ajouter une méthode affiche() qui est appelée par la surcharge externe de <<

2) celle-ci est très laide mais syntaxiquement correcte, donc acceptée : rendre public les deux attributs **x** et **s**

1.3) Quel est le résultat de l'exécution de ce programme après correction des erreurs.

1.3.1) JUSTIFIER chaque valeur affichée par l'exécution de la **ligne 30**. Rappel : les règles de priorité entre opérateurs s'appliquent aussi aux opérateurs surchargés.

la règle d'associativité gauche-droite de l'opérateur + s'applique, ce qui correspond à l'évaluation des sous-expressions suivantes :

```
[W]: ( ( ( ob1 + ob3 ) + ob4 ) + ob2)
[B]: ( ( ( ob2 + ob3 ) + ob4 ) + ob1)
[Y]: ( ( ( ob2 + ob4 ) + ob3 ) + ob1)
[S]: ( ( ( ob1 + ob4 ) + ob3 ) + ob2)
```

[All] : quel que soit l'ordre des opérandes on obtient toujours **x: 100** pour l'addition des entiers 10, 20, 30 et 40. Par contre la concaténation des **string** dépend de l'ordre des opérandes, ce qui donne :

```
[W]: s: 10304020
[B]: s: 20304010
[Y]: s: 20403010
[S]: s: 10403020
```

1.3.2) JUSTIFIER la valeur affichée par l'exécution de la **ligne 32**

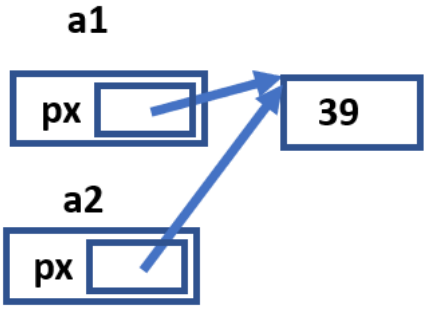
[All] : la valeur affichée pour la variable de classe c est **7** car elle est initialisée à 0 ligne 14 et est incrémentée à chaque appel du constructeur (ligne 6). Cela correspond au nombre d'instances créées, soit **4** pour les déclarations de la ligne 28 et **3** pour la valeur renvoyée par les **3** sous-expressions d'addition à la ligne 30.

2) (6 pts) Analyse de code :

Le code **ex2.cc** compile avec succès en C++11 sans produire de warning. Pourtant, après l'affichage de quelques lignes, l'exécution se termine par un message signalant un problème.

```
1   #include <iostream>
2
3   class A {
4   public:
5     A(int new_x): px(new int(new_x)) {}
6     ~A()           {delete px; }
7     int f() const {return 1 + *px; }
8     int* getpx()   {return px; }
9
10  protected:
11    int *px;
12  };
13
14  class B : public A {
15  public:
16    B(): A(5) {}
17    ~B() {}
18    int f() const {return 2 + *px; }
19  };
20
21  int main() {
22    A a1(39);
23    A a2(a1);
24
25    std::cout << *a2.getpx() << std::endl;
26
27    (*a2.getpx()) += 2;
28    std::cout << *a2.getpx() << std::endl;
29
30    std::cout << ++(*a1.getpx()) << std::endl;
31
32    B b;
33    A a3 = b;
34    std::cout << a3.f() << std::endl;
35
36    return 0;
37  }
```

2.1) **ligne 24** : dessiner l'état des variables **a1** et **a2** avec la représentation « boîte et flèche » après leur initialisation :



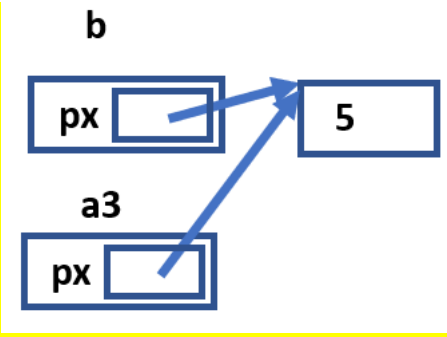
[W] remplacer 39 par : 37 pour [B], 36 pour [Y], 38 pour [S]

2.2) **ligne 25** : JUSTIFIER la valeur affichée
a2 possède la même valeur que **a1** pour son attribut **px** ; c'est cette adresse qui est renvoyée. Comme on fait une indirection sur cette adresse la valeur de l'expression qui est affichée est l'entier mémorisé à cette adresse c'est-à-dire **[W] : 39 , [B] : 37, [Y] : 36, [S] : 38**.

2.3) **ligne 28** : JUSTIFIER la valeur affichée
D'abord, ligne 27, L'adresse mémorisée dans **px** de **a2** est utilisée pour accéder et modifier le contenu de la mémoire qui est alors augmenté de 2 et prend la valeur **[W] : 41 , [B] : 39, [Y] : 38, [S] : 40**. C'est cette nouvelle valeur qui est affichée à la ligne 28

2.4) **ligne 30** : JUSTIFIER la valeur affichée
L'adresse mémorisée dans **px** de **a1** est utilisée avec l'opérateur d'indirection pour désigner l'entier manipulé aux questions précédentes qui se trouve pré-incrémenté avant d'être affiché. On obtient : **[W] : 42 , [B] : 40, [Y] : 39, [S] : 41**.

2.5) **ligne 34** : dessiner l'état des variables **b** et **a3** avec la représentation « boîte et flèche » après leur initialisation et JUSTIFIER la valeur affichée

[W,S] 

remplacer 5 par : 3 pour [B], 7 pour [Y]

a3 étant de la classe de base c'est la méthode **f**() de cette classe qui est appelé et qui évalue l'expression 1+ *px ; on obtient l'affichage de **[W,S] : 6, [B] :4, [Y] :8 .**

2.6) proposer une explication concernant l'obtention d'un message d'erreur après l'exécution des affichages précédents

[All] L'appel du destructeur de **a3** libère la mémoire pointée par **px**, ce qui se passe normalement. L'erreur survient dès l'appel du destructeur de **b** qui appelle d'abord celui de la classe B, qui ne fait rien, avant d'appeler automatiquement le destructeur de la superclasse A. C'est là que le problème de double-libération de mémoire est détecté quand ce destructeur de A veut libérer une seconde fois l'espace pointé par **px**.

Si main() n'avait pas contenu les lignes 32-34, il y aurait eu le même problème avec l'appel des destructeurs de a2 (d'abord) puis de a1.

3) (6 pts) une petite dose de vector

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

```
1   #include <iostream>
2   #include <vector>
3
4   using namespace std;
5
6   class MyClass {
7   public:
8       MyClass(int id) : id_(id) {
9           cout << "Constructeur pour ID: " << id_ << endl;
10      }
11      ~MyClass() {
12          cout << "Destructeur pour ID: " << id_ << endl;
13      }
14      int getID() const { return id_; }
15  private:
16      int id_;
17  };
18
19  int main() {
20      vector<MyClass *> myVector;
21
22      for (int i = 1; i <= 3; ++i) {
23          myVector.push_back(new MyClass(i));
24      }
25
26      for (MyClass *ptr : myVector) {
27          cout << "ID: " << ptr->getID() << endl;
28      }
29      myVector.clear();
30
31      return 0;
32  }
```

3.1) JUSTIFIER les affichages obtenus par son exécution **[All]**

La première boucle appelle trois fois le constructeur de la classe qui affiche

```
Constructeur pour ID: 1
Constructeur pour ID: 2
Constructeur pour ID: 3
```

C'est l'adresse des instances qui est mémorisée dans le vector **myVector** car il s'agit d'une allocation dynamique avec l'opérateur **new**. C'est pourquoi la seconde boucle obtient un pointeur par élément du vector et utilise l'opérateur **->** pour appeler la méthode **getID**. On obtient l'affichage:

```
ID: 1
ID: 2
ID: 3
```

Il n'y a AUCUN appel du destructeur et donc aucun affichage supplémentaire comme précisé ci-dessous.

3.2) PRECISER l'impact de la **ligne 29** ; comment appelle-t-on le problème qui pourrait en résulter pour un programme plus ambitieux ayant les lignes 22-29 dans une boucle `while(true)` ?

[All] la ligne 29 vide seulement le **vector** mais sans libérer la mémoire pointée par chaque pointeur ; donc le destructeur de **MyClass** n'est jamais appelé. Ce problème s'appelle une ***fuite de mémoire*** car avec l'appel de **clear()** le programme perd les addresses et il n'est plus du tout possible de les libérer plus tard. Cela causera un échec d'une demande future d'allocation dynamique si les lignes 22-29 sont dans une boucle infinie.

4) (5 pts) un petit héritage : soit le fichier **ex4.cc** dont le code est listé ci-dessous
Ce code **ex4.cc** compile sans warning avec -std=c++11 -Wall. , Son exécution se termine normalement.

```
1   #include <iostream>
2   #include <string>
3
4   using namespace std;
5
6   class Pet {
7   public:
8       Pet(const string& name = "animal"){
9           cout << "Created " << name << "." << endl;
10      }
11      virtual ~Pet() {}
12      virtual void makeSound(){
13          cout << "Generic pet sound" << endl;
14      }
15  };
16
17  class Dog : public Pet {
18  public:
19      Dog() : Pet("dog") {}
20      void makeSound() override { cout << "Woof!" << endl; }
21  };
22
23  int main() {
24      Dog myDog;
25      Pet pet(myDog);
26      pet.makeSound();
27      return 0;
28  }
```

4.1) JUSTIFIER tout ce qui est affiché par son exécution **[All]**

Ligne 24 la déclaration de l'instance **myDog** produit l'appel du constructeur de la classe **Pet** avec le paramètre `"dog"`, ce qui produit l'affichage de :
`Created dog`

Ligne 25 la déclaration de l'instance **pet** utilise le ***constructeur de copie par défaut*** de la classe **Pet**. Celui-ci ne fait rien et, en tout cas, aucun affichage car le constructeur de la classe Pet n'est pas appelé par le constructeur de copie.

Ligne 26 la méthode **makeSound** de la classe de base **Pet** est appelée car **pet** est une instance de cette classe **Pet**, ce qui produit l'affichage de :
`Generic pet sound`

5) (6 pts) un autre héritage : soit le fichier **ex5.cc** dont le code est listé ci-dessous
Ce code **ex5.cc** compile sans warning avec -std=c++11 -Wall. , Son exécution se termine normalement.

```
1   #include <iostream>
2   #include <array>
3   using namespace std;
4
5   class Animal {
6   public:
7       Animal(string name) : name(name) {}
8       virtual ~Animal() {}
9       virtual void speak() const = 0 ;
10      void greet() const { cout << "Hello, I am " << name << endl; }
11      const string& getName() const { return name; }
12  private:
13      string name;
14  };
15
16  class Lion : public Animal {
17  public:
18      Lion() : Animal("Leo") {}
19      void speak() const { cout << "Roar" << endl; }
20  };
21
22  class Tiger : public Animal {
23  public:
24      Tiger() : Animal("Simba") {}
25      void speak() const { cout << "Moan" << endl; }
26      void greet() const { cout << "Greetings from a tiger" << endl;  }
27  };
28
29  int main() {
30    Lion  x;
31    Tiger y;
32
33    array<Animal*,2> tab={&x,&y};
34    for( auto p : tab){
35          p->greet();
36          p->speak();
37    }
38  }
```

5.1) JUSTIFIER tout ce qui est affiché par son exécution

Les constructeurs des lignes 30 et 31 initialisent seulement l'attribut **name** de la superclasse **Animal**. Ensuite l'array **tab** de 2 pointeurs **Animal*** est initialisé avec les adresses des instances **x** et **y**. Cela permet de bénéficier du ***polymorphisme*** pour la méthode ***virtuelle*** **speak** car elle est déclarée avec **virtual**, contrairement à la méthode **greet** qui n'est donc ***pas virtuelle***. On obtient donc les affichages suivants :

[W]	[B]	[Y]	[S]
Hello, I am Leo Roar Hello, I am Simba Moan	Hello, I am Leon Roar Hello, I am Simbax Moan	Hello, I am Leox Roar Hello, I am Simbad Moan	Hello, I am Leonidas Roar Hello, I am Simbar Moan

Remarque concernant la notation : les affichages **Roar** et **Moan** sont considérées comme faux si la JUSTIFICATION ne précise pas le contexte **polymorphique**. En particulier la seule indication d'une « *redéfinition* » de **speak** dans les classes dérivées n'est pas suffisante et n'est pas acceptée comme réponse correcte car la boucle travaille avec des pointeurs de type **Animal***.

6) (6 pts) héritage multiple : soit le fichier **ex6.cc** dont le code est listé ci-dessous
Ce code **ex6.cc** compile sans warning avec -std=c++11 -Wall. , Son exécution se termine normalement.

```
1   #include <iostream>
2   using namespace std;
3
4   class A {
5   public:
6         A() { cout << "Constructeur A" << endl; }
7         virtual void print() const { cout << "Affichage A" << endl; }
8   };
9
10  class B : virtual public A {
11  public:
12        B() { cout << "Constructeur B" << endl; }
13        void print() const override { cout << "Affichage B" << endl; }
14  };
15
16  class C : virtual public A {
17  public:
18        C() { cout << "Constructeur C" << std::endl; }
19        void print() const override { cout << "Affichage C" << endl; }
20  };
21
22  class D : public B, public C {
23  public:
24        D() { cout << "Constructeur D" << endl; }
25        void print() const override { cout << "Affichage D" << endl; }
26  };
27
28  int main() {
29        D* d1 = new D();
30        A d2 = *d1;
31        d2.print();
32        return 0;
33  }
```

6.1) JUSTIFIER tout ce qui est affiché par son exécution

[All] L'héritage est *virtuel*, ce qui veut dire que le constructeur de la classe A est appelé une seule fois et en premier, puis on appelle les constructeurs des superclasses dans l'ordre indiqué ligne 22, B puis C, et enfin le constructeur de la classe D. Pour l'allocation dynamique de l'instance pointée par **d1** à la ligne 29, on obtient l'affichage :

```
Constructeur A
Constructeur B
Constructeur C
Constructeur D
```

Ensuite un *constructeur de copie par défaut* est appelé pour l'instance **d2** de la classe A, ce qui n'affiche rien car le constructeur de la classe A n'est pas appelé. Ensuite, l'appel de la méthode **`print`** sur l'instance **d2** de A appelle la méthode de la classe A qui affiche :

```
Affichage A
```

6.2) Supposons maintenant que la ligne 25 est en commentaire ; qu'en déduisez-vous ?

On obtient une erreur de compilation car le C++ exige que **`print`** soit redéfinie explicitement dans la classe D à cause de l'ambigüité de choix dans une classe intermédiaire comme B ou C.

Cette erreur est signalée même si aucun appel à cette méthode n'est fait dans le code pour une instance de la classe D, comme c'est le cas ici car **d2** est de type A. Remarque : Il n'y aurait pas d'erreur de compilation si aucune ou une seule des deux classes, B ou C, redéfinssait **`print`**.