université PARIS DIDEROT PARIS 7

EXAMEN | LOAV — C++ | 11/01/2018 15^{30}–17^{30}

Aucun document ou support autre que le sujet ou les copies d'examen n'est autorisé.
(la copie ou les brouillons du voisin ne sont pas des supports autorisés).
Éteignez impérativement vos mobiles.

1 Exercice

On souhaite pouvoir écrire le code suivant :

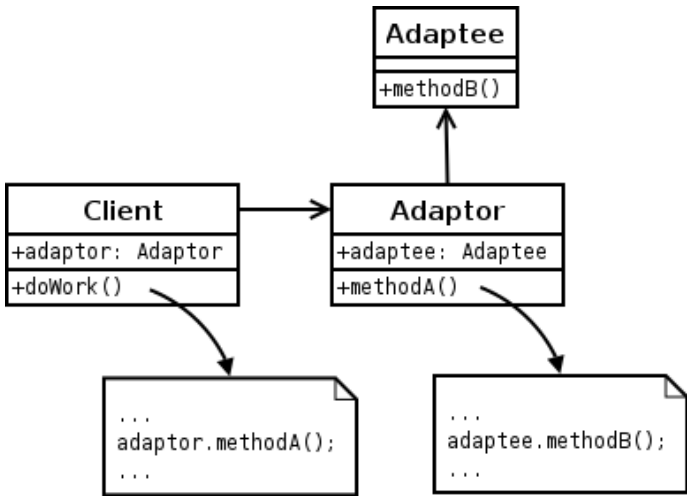
```
int main() {
  Boolean b(false), c(true), d = Boolean::TRUE;
  Boolean a(true);
  a = (b+c)*-d+Boolean::TRUE;
  std::cout << a << std::endl;
  if (a) std::cout << "YES" << std::endl;
}
```

qui correspond à l'utilisation d'un type Booléen (supposé plus «solide» que le type `bool` du C++) pour lequel l'arithmétique utilise `+` pour le OU, `*` pour le ET et `-` pour la négation. On souhaite que ce type se comporte de façon raisonnable (par exemple impossible de créer des variable non initialisées, possibilité d'affectation, etc).

1. Quels sont les constructeurs nécessaires ? Expliquer pourquoi et donner leurs prototypes.
2. De quelle nature les variables `Boolean::TRUE` et `Boolean::FALSE` sont elles ? Comment seront-elles déclarées/définies (donner la portion de code utile).
3. Quels sont les conversions entre types nécessaires au bon fonctionnement du programme proposé ci-dessus ? Donner leurs prototypes.
4. Quels sont les opérateurs nécessaires au bon fonctionnement du programme proposé ci-dessus ? Donner leurs prototypes.
5. Définir la classe `Boolean` et les fonctions nécessaires à la bonne compilation et au bon fonctionnement du code proposé ci-dessus.

2 Exercice

Soit le diagramme UML représentant le pattern adaptateur :



Implémenter ce pattern en C++ sous la forme d'un programme complet (définition des classes et exemple d'utilisation du pattern dans le `main`).

3 Problème

La phylogénie nous indique que les Bovins (qui ont un cuir d'une certaine épaisseur) et les Ovins (qui sont couverts de laine d'une certaine longueur) sont des Ruminants (qui ruminent) eux-même des Mammifères (qui allaitent leurs petits). Parmi les Bovins on trouve les *Bos Taurus* dont la représentante la plus connue est Marguerite, jolie vache laitière de 564kg et élevée par Marcel et Germaine Delapeau, fermiers à Brive-la-gaillarde. Pour information, les Vaches sont des *Bos Taurus* femelles et les Taureaux des *Bos Taurus* mâles.

1. Comment traduit-on en UML une telle spécification ?
2. Placer dans les classes identifiées les caractéristiques/opérations sous-entendues par la spécification.
3. Traduire chaque classe UML en une classe C++ correspondante avec attributs et méthodes. Prendre soin de spécifier tous les modificateurs adéquats, nécessaires ou utiles (const, etc).
4. Écrire un programme C++ permettant de créer une version numérique de la charmante Marguerite et de ses propriétaires et d'afficher leurs caractéristiques.
5. Écrire une usine (factory) permettant de créer un *Bos Taurus* de sexe, nom et race quelconques (sous la forme d'attributs). Modifier la classe Vache en conséquence. Modifier la construction de Marguerite afin d'obtenir une Vache de race Limousine.
6. Si l'on souhaite obliger l'utilisateur à passer par l'usine pour obtenir une Vache, que doit-on modifier ?
7. Si l'on décide de distinguer les races de Bœufs à l'aide de classes différentes et non plus d'attributs, qu'est-ce que cela change ? Écrire les classes Limousine et Normande.
8. Sachant qu'une Limousine fait « Meuheuh » et une Normande fait « Meeeuuuhhh » juste après avoir mangé, implanter ce mécanisme dans les classes adéquates. Attention, les Ovins eux, ne font aucun bruit particulier après avoir mangé...

4 Problème

On souhaite construire un système de type permettant de manipuler des quantités physiques, c'est-à-dire associées à des unités : 12,5m ce n'est pas ma meme chose que 12,5s.

1. Soit la classe :

```
class Longueur {
private:
  double q;
public:
  Longueur(double q) : q(q) {}
  double getValue() const { return q; }
};
std::ostream &operator<<(std::ostream &os,const Longueur &l) {
  return os << l.getValue() << "m";
}
```

On imagine la définition des classes `Temps` et `Vitesse`, définir les opérateurs permettant d'obtenir une vitesse par division d'une longueur et d'un temps et une longueur par multiplication d'une vitesse et d'un temps de sorte que l'on puisse écrire :

```
  Longueur l(100);
  Temps t(50);
  Vitesse v = l/t;
  std::cout << l << std::endl;
  std::cout << t << std::endl;
  std::cout << v << std::endl;
  Temps t2(20);
  Longueur l2 = v*t2;
  std::cout << l2 << std::endl;
```

et obtenir

```
100m
50s
2m/s
40m
```

2. À l'aide d'un héritage, factoriser les classes `Longueur`, `Temps` et `Speed` de façon à éviter la redéfinition du champ, etc. La classe de factorisation s'appellera `Quantite`. On songera à factoriser autant que possible l'opérateur d'affichage d'une quantité et de son unité associée.

3. La factorisation allège largement la définition des classes de quantités physiques, mais il reste que pour chaque type d'unité on doit définir une classe et les opérateurs associés (il faut imaginer définir des accélérations, des surfaces, etc). On se propose de construire un système de type de sorte qu'une quantité soit typée par une longueur en dimension quelconque et un temps en dimension quelconque. Ainsi une accélération est typée par $m^1.s^{-2}$, une longueur par $m^1.s^0$, un temps par $m^0.s^1$, etc. Une quantité dans ce système est donc une valeur associée à une dimension dans l'espace et une dimension dans le temps $m^S.s^T$. On notera qu'une longueur divisée par un temps, produit une vitesse, qu'une vitesse divisée par un temps est une accélération, on a donc des équations sur les unités de sorte que $m^S.s^T \times m^{S'}.s^{T'} = m^{S+S'}.s^{T+T'}$, ou $m^S.s^T + m^S.s^T = m^S.s^T$, etc. C'est ce que l'on appelle l'analyse dimensionnelle et qui permet d'assurer l'homogénéité de calculs sur des quantités physiques

   À l'aide d'un template à deux arguments (les dimensions en espace et en temps) définir une quantité physique, puis définir l'opérateur permettant d'afficher une quantité physique, puis des opérateurs permettant de construire n'importe quelle quantité physique à l'aide de la division ou la multiplication, de sorte que l'on puisse écrire :

```
typedef PhysicalQuantity<1,0> L;
typedef PhysicalQuantity<0,1> T;
typedef PhysicalQuantity<1,-1> S;
typedef PhysicalQuantity<0,0> N;

L lll(500);
std::cout << lll << std::endl;
T ttt(50);
std::cout << ttt << std::endl;
S sss = lll/ttt;
std::cout << sss << std::endl;
PhysicalQuantity<1,-2> aaa = sss/ttt;
std::cout << aaa << std::endl;
N n = lll/lll;
std::cout << n << std::endl;
double q = lll/lll;
std::cout << q << std::endl;
std::cout << "-------------" << std::endl;
```

   qui produit :

```
500m
50s
10ms^-1
0.2ms^-2
1
1
```