Université de Nice-Sophia Antipolis — POLYTECH
ELSE4-FISE — 2023–2024

Examen de C++

Durée : 1h00
Aucun document autorisé
Mobiles interdits

1 Étudiants/Enseignants

Un étudiant est défini par son nom et une liste de noms de cours qu'il suit. Un enseignant est défini par son nom et le nom du cours (un seul) qu'il enseigne. On veut pouvoir exécuter le code suivant :

```
std::vector<std::string> mat1 = {"C++", "TNS", "Anglais"};
std::vector<std::string> mat2 = {"Auto", "Élec Analog"};
std::vector<Personne*> liste = {
  new Étudiant("Paul", mat1),
  new Étudiant("Nathalie", mat2),
  new Enseignant("Vincent", "C++")
};

for (Personne *p : liste)
  std::cout << *p << std::endl;
```

qui produit le résultat suivant :

```
Paul : C++ TNS Anglais
Nathalie : Auto Élec Analog
Vincent (enseignant) : C++
```

► 1. Écrivez les classes nécessaires à la programmation du code précédent. Vous expliquerez clairement les mécanismes mis en jeu.

```
#include <string>
#include <vector>

/*
 * Classe mère des classes Étudiant et Enseignant. Elle factorise
 * le nom de des étudiants et des enseignants.
 */
class Personne {
protected:
    std::string nom;
public:
  Personne(const std::string& n) : nom(n) {}

  virtual std::string toString() const {
    return this->nom;
  }
```



```
  /*
   *   Dans cette méthode, la méthode toString sera appliquée
   *   par liaison dynamique sur la Personne désignée par le paramètre
   *   p à l'exécution au moment de son application
   */
  friend std::ostream &operator<<(std::ostream &f, const Personne &p) {
    return f << p.toString();
  }
};

class Étudiant : public Personne {
protected:
  std::vector<std::string> matières; // spéficique à un étudiant
public:
    Étudiant(const std::string& n, std::vector<std::string> &m) :
      Personne(n), matières(m)
   {}

  /*
   * redéfinition de toString, réutilisation du toString de la classe mère
   */
  virtual std::string toString() const override {
    std::string s=Personne::toString() + " : ";
    for (std::string m : this->matières)
      s+=m+ " ";
    return s;
  }
};

class Enseignant : public Personne {
protected:
    std::string cours; // spéficique à un enseignant
public:
    Enseignant(const std::string& n,  const std::string& c) :
      Personne(n), cours(c)
    {}

  /*
   * redéfinition de toString, réutilisation du toString de la classe mère
   */
  virtual std::string toString() const override {
    return Personne::toString() + " (enseignant) : " + this->cours;
  }
};
```

.....

2 Division

► 2. Écrivez la classe `DivOException`, héritière de la classe `std::exception`, avec la méthode `what` qui renvoie le message *Exception : division par 0.*

```
/**
 * cette classe représente une exception émise
```

```
 * lors du division par 0
 */
class Div0Exception : public std::exception {
public:
  // redéfinition de what
  virtual const char* what() const noexcept override {
    return "Exception : division par 0";
  }
};
```

.....

▶ 3. Écrivez la fonction `diviser` qui renvoie la division de ses deux paramètres, `numérateur` et `dénominateur` de type **double**. Si le dénominateur est égal à 0.0, la fonction émettra l'exception `Div0Exception`.

```
/*
 * Rôle : renvoie numérateur/dénominateur
 * Exception : émet l'exception Div0Exception si dénominateur est égal à 0
 */
double diviser(const double numérateur, const double dénominateur) {
  if (dénominateur==0) throw Div0Exception();
  return numérateur/dénominateur;
}
```

.....

▶ 4. Écrivez la fonction `main` qui lit deux **double**, `x` et `y`, sur l'entrée standard et écrit sur la sortie standard le résultat de l'appel de fonction `diviser`, ou le message de l'exception sur la sortie d'erreur standard.

```
int main() {
  int x, y;

  std::cout << "x = "; std::cin >> x;
  std::cout << "y = "; std::cin >> y;
  try {
    std::cout << diviser(x,y) << std::endl;
    return EXIT_SUCCESS;
  }
  catch (const Div0Exception &e) {
    std::cerr << e.what() << std::endl;
    return EXIT_FAILURE;
  }
}
```

.....