TRAVAUX PARATIQUES

IG /IUT

MEMBRES DU GROUPE :

1-ARAYE Max Donald

2-KINDOHO Priscille

3-KORA ZIME Lawal

4-OGOUREMI Eustache

5-SEGBOZO O'nel

2 Construction d'une classe en langage C++

1- Le fait de limiter le choix d'attributs d'une entité du monde réel à un sous-ensemble fini en POO est appelé encapsulation.

2- Dans la POO, Un objet est une instance d'une classe, c'est à dire un emplacement mémoire adressable dont le type est celui de la classe .

Une classe est une structure de données définissant les attributs et les méthodes communes à ses instances.

Une instance de classe est une manifestation spécifique de la structure et du comportement définis par une classe dans le monde reel du programme en cours d'exécution .

3- Voici les classes demandées en utilisant le modèle trivial :

```
#include <iostream>

#include <string>


class Personne {

public:

   // Attributs communs

   std::string nom;

   std::string prenom;

   int age;

   char sexe;

   std::string lieuDeNaissance;

   std::string numeroTelephone;


   // Fonctionnalités communes

   void manger() {

      std::cout << nom << " " << prenom << " mange." << std::endl;
```

```
    }

    void afficher() {
        std::cout << nom << " " << prenom << " affiche quelque chose." << std::endl;
    }

    void toString() {
        std::cout << "Nom: " << nom << ", Prénom: " << prenom << ", Age: " << age
                  << ", Sexe: " << sexe << ", Lieu de naissance: " << lieuDeNaissance
                  << ", Numéro de téléphone: " << numeroTelephone << std::endl;
    }

    void dormir() {
        std::cout << nom << " " << prenom << " dort." << std::endl;
    }

    // Constructeurs et destructeur
    Personne() {
        std::cout << "Création d'une instance de Personne par le constructeur par défaut." << std::endl;
    }

    Personne(const Personne& other) {
        nom = other.nom;
        prenom = other.prenom;
        age = other.age;
```

```
        sexe = other.sexe;
        lieuDeNaissance = other.lieuDeNaissance;
        numeroTelephone = other.numeroTelephone;
        std::cout << "Création d'une instance de Personne par le constructeur par copie." << std::endl;
    }

    Personne(std::string _nom, std::string _prenom, int _age, char _sexe,
             std::string _lieuDeNaissance, std::string _numeroTelephone) :
        nom(_nom), prenom(_prenom), age(_age), sexe(_sexe),
        lieuDeNaissance(_lieuDeNaissance), numeroTelephone(_numeroTelephone) {
        std::cout << "Création d'une instance de Personne par le constructeur complet." << std::endl;
    }

    virtual ~Personne() {
        std::cout << "Destruction d'une instance de Personne." << std::endl;
    }
};

class Etudiant : public Personne {
public:
    // Attributs spécifiques à l'étudiant
    std::string universite;
    std::string faculte;
    std::string filiere;
```

```
    int anneeEtude;
    std::string numeroMatricule;
    std::string adresseEmail;

    // Fonctionnalités spécifiques à l'étudiant
    void etudier() {
        std::cout << nom << " " << prenom << " étudie." << std::endl;
    }

    void evaluer() {
        std::cout << nom << " " << prenom << " se fait évaluer." << std::endl;
    }

    void distraire() {
        std::cout << nom << " " << prenom << " se distrait." << std::endl;
    }

    // Constructeurs et destructeur
    Etudiant() : Personne() {
        std::cout << "Création d'une instance d'Etudiant par le constructeur par défaut." << std::endl;
    }

    Etudiant(const Etudiant& other) : Personne(other) {
        universite = other.universite;
        faculte = other.faculte;
        filiere = other.filiere;
```

```
        anneeEtude = other.anneeEtude;
        numeroMatricule = other.numeroMatricule;
        adresseEmail = other.adresseEmail;
        std::cout << "Création d'une instance d'Etudiant par le constructeur par copie." << std::endl;
    }

    Etudiant(std::string _nom, std::string _prenom, int _age, char _sexe,
             std::string _lieuDeNaissance, std::string _numeroTelephone,
             std::string _universite, std::string _faculte, std::string _filiere,
             int _anneeEtude, std::string _numeroMatricule, std::string _adresseEmail) :
        Personne(_nom, _prenom, _age, _sexe, _lieuDeNaissance, _numeroTelephone),
        universite(_universite), faculte(_faculte), filiere(_filiere),
        anneeEtude(_anneeEtude), numeroMatricule(_numeroMatricule), adresseEmail(_adresseEmail) {
        std::cout << "Création d'une instance d'Etudiant par le constructeur complet." << std::endl;
    }

    ~Etudiant() override {
        std::cout << "Destruction d'une instance d'Etudiant." << std::endl;
    }
};

class Enseignant : public Personne {
public:
```

```
    // Attributs spécifiques à l'enseignant
    std::string specialites;
    std::string matieresEnseignees;
    std::string emploiDuTemps;

    // Fonctionnalités spécifiques à l'enseignant
    void enseigner() {
        std::cout << nom << " " << prenom << " enseigne." << std::endl;
    }

    void conseiller() {
        std::cout << nom << " " << prenom << " conseille." << std::endl;
    }

    void orienter() {
        std::cout << nom << " " << prenom << " oriente." << std::endl;
    }

    void evaluerTravaux() {
        std::cout << nom << " " << prenom << " évalue les travaux de ses apprenants." << std::endl;
    }

    // Constructeurs et destructeur
    Enseignant() : Personne() {
        std::cout << "Création d'une instance d'Enseignant par le constructeur par défaut." << std::endl;
```

```
    }

    Enseignant(const Enseignant& other) : Personne(other) {
        specialites = other.specialites;
        matieresEnseignees = other.matieresEnseignees;
        emploiDuTemps = other.emploiDuTemps;
        std::cout << "Création d'une instance d'Enseignant par le constructeur par copie." << std::endl;
    }

    Enseignant(std::string _nom, std::string _prenom, int _age, char _sexe,
               std::string _lieuDeNaissance, std::string _numeroTelephone,
               std::string _specialites, std::string _matieresEnseignees,
               std::string _emploiDuTemps) :
        Personne(_nom, _prenom, _age, _sexe, _lieuDeNaissance, _numeroTelephone),
        specialites(_specialites), matieresEnseignees(_matieresEnseignees),
        emploiDuTemps(_emploiDuTemps) {
        std::cout << "Création d'une instance d'Enseignant par le constructeur complet." << std::endl;
    }

    ~Enseignant() override {
        std::cout << "Destruction d'une instance d'Enseignant." << std::endl;
    }
};
```

```
class Administratif : public Personne {
public:
    // Attributs spécifiques à l'administratif
    std::string qualifications;
    std::string tachesARealiser;
    std::string poste;
    std::string emploiDuTemps;

    // Fonctionnalités spécifiques à l'administratif
    void conseiller() {
        std::cout << nom << " " << prenom << " conseille." << std::endl;
    }

    void realiserTaches() {
        std::cout << nom << " " << prenom << " réalise ses tâches." << std::endl;
    }

    // Constructeurs et destructeur
    Administratif() : Personne() {
        std::cout << "Création d'une instance d'Administratif par le constructeur par défaut." << std::endl;
    }

    Administratif(const Administratif& other) : Personne(other) {
        qualifications = other.qualifications;
        tachesARealiser = other.tachesARealiser;
        poste = other.poste;
```

```
        emploiDuTemps = other.emploiDuTemps;
        std::cout << "Création d'une instance d'Administratif par le constructeur par copie." << std::endl;
    }

    Administratif(std::string _nom, std::string _prenom, int _age, char _sexe,
                  std::string _lieuDeNaissance, std::string _numeroTelephone,
                  std::string _qualifications, std::string _tachesARealiser,
                  std::string _poste, std::string _emploiDuTemps) :
        Personne(_nom, _prenom, _age, _sexe, _lieuDeNaissance, _numeroTelephone),
        qualifications(_qualifications), tachesARealiser(_tachesARealiser),
        poste(_poste), emploiDuTemps(_emploiDuTemps) {
        std::cout << "Création d'une instance d'Administratif par le constructeur complet." << std::endl;
    }

    ~Administratif() override {
        std::cout << "Destruction d'une instance d'Administratif." << std::endl;
    }
};

class Technicien : public Personne {
public:
    // Attributs spécifiques au technicien
    std::string qualifications;
    std::string emploiDuTemps;
```

```
    // Fonctionnalités spécifiques au technicien
    void realiserTaches() {
        std::cout << nom << " " << prenom << " réalise les tâches qui lui sont attribuées." << std::endl;
    }

    // Constructeurs et destructeur
    Technicien() : Personne() {
        std::cout << "Création d'une instance de Technicien par le constructeur par défaut." << std::endl;
    }

    Technicien(const Technicien& other) : Personne(other) {
        qualifications = other.qualifications;
        emploiDuTemps=other.emploiDuTemps ;
     std::cout << "Création d'une instance d'Administratif par le constructeur par copie." << std::endl;
}
Technicien(std::string _nom, std::string _prenom, int _age, char _sexe,
                std::string _lieuDeNaissance, std::string _numeroTelephone,
                std::string _qualifications,std ::string_emploiDuTemps) :
        Personne(_nom, _prenom, _age, _sexe, _lieuDeNaissance, _numeroTelephone),
      qualifications(qualifications) , emploiDuTemps(_emploiDuTemps) {
        std::cout << "Création d'une instance d'Administratif par le constructeur complet." << std::endl;
    }
```

```
~Technicien() override {
        std::cout << "Destruction d'une instance Technicien." << std::endl;
    }
};
```

4- Une méthode qui permet de créer une instance d’un objet est appelée constructeur. L'identifiant d'une telle méthode est le nom de la classe.

Elle ne renvoie pas de résultat car elle est chargée de l'initialisation de l'objet.

5- Le constructeur par défaut d'une classe est une méthode appelée lorsqu'une instance est créée sans paramètres. Il est utile pour fournir des valeurs par défaut aux attributs.

6- Pour représenter un individu inconnu, il est possible utiliser des valeurs par défaut pour les attributs ou créer une méthode spécifique, par exemple `creerIndividuInconnu()`.

7- Un constructeur par copie crée une nouvelle instance en copiant les valeurs d'une instance existante.

Il est délicat car il peut entraîner des problèmes de gestion de mémoire.

Pour l'interdire, on peut le déclarer comme privé sans le définir.

8- Une méthode définie plusieurs fois dans une classe avec des signatures différentes est une surcharge de méthode.

Cela est nécessaire pour permettre à la méthode d'accepter différents types de paramètres.

9- VOIR Q3 (Classes)

10- Le destructeur d'une classe est appelé lorsqu'une instance est détruite. Il doit être virtuel pour permettre une gestion correcte de la mémoire.

Il n'y a besoin que d'un seul destructeur par classe car il est appelé automatiquement lors de la destruction de l'objet.

11- VOIR Q3 (Classes)

12- VOIR Q3

13- Il est nécessaire de détruire soi-même un objet créé dynamiquement pour libérer la mémoire allouée avec `new`.

14- En commentant les lignes 10 et 11, et en décommentant la ligne 13, le résultat du programme restera le même, car le constructeur par défaut (ligne 10 n'est pas utilisé dans le code fourni. La ligne 13 offre simplement une nouvelle option pour créer un objet avec un paramètre d'âge.

Commenter ou décommenter la ligne 15 n'aura pas non plus d'impact sur le résultat du programme, car elle représente un constructeur avec une liste d'initialisation complète qui n'est pas utilisé dans le code fourni

15- L'utilisation du constructeur avec liste d'initialisation permet de généraliser la définition de plusieurs constructeurs en spécifiant les valeurs des attributs directement à la création.

16- Si la ligne 7 du programme est commentée, cela supprimerait la déclaration de la classe Toto dans son ensemble. Cela conduirait à une erreur de compilation, car le compilateur ne reconnaîtrait pas la classe Toto lorsqu'elle est utilisée dans le reste du programme.

Les messages du compilateur indiquent une erreur du type "undeclared identified 'Toto' indiquant qu'il ne peut pas trouver la déclaration de la classe 'Toto' .

17- L'initialisation des variables statiques doit être faite en dehors de la classe.

18- Le modèle trivial peut poser des problèmes en termes de duplication de code et de maintenance, notamment lorsqu'il y a des fonctionnalités communes entre les classes.

19- L'héritage est le mécanisme de la POO qui permet d'éliminer la redondance. Vous pouvez créer une classe de base `Personne` avec les attributs et méthodes communs, puis hériter de cette classe pour chaque type d'individu.

20- VOIR Q3.

3-MISE EN ŒUVRE DE L'ENCAPSULATION

1.) Pour rendre les attributs d'une classe accessibles de n'importe où, il faut les déclarer comme des attributs publics en dehors des méthodes, généralement en haut de la classe. Cependant, cela peut entraîner des failles de sécurité, car cela expose directement les données internes de la classe, permettant une modification non contrôlée et augmentant le couplage entre les différentes parties du code.

2.) Déclarer les attributs d'une classe dans la zone protected offre l'avantage de permettre l'accès à ces attributs aux classes dérivées (sous-classes). Cela favorise l'héritage et l'extension des fonctionnalités sans exposer directement les attributs aux classes externes.

Cependant, cela peut présenter l'inconvénient de rendre les attributs accessibles aux classes dérivées, ce qui peut entraîner une dépendance accrue entre les classes. De plus, cela peut rendre la gestion de l'encapsulation plus complexe, car la visibilité protected signifie que les attributs sont accessibles non seulement à la classe elle-même, mais aussi à ses sous-classes.

3.) L'emploi systématique de la zone private pour l'ajout des attributs d'un objet est souvent recommandé pour encapsuler les données de manière sécurisée. Cela permet de restreindre l'accès direct aux attributs depuis l'extérieur de la classe, favorisant ainsi l'encapsulation et la modularité du code. En limitant l'accès, on réduit les risques de manipulation incorrecte des données et on encourage l'utilisation de méthodes publiques pour interagir avec ces attributs, ce qui peut aider à maintenir un code plus robuste et compréhensible.

4.) L'exemple d'un objet qui nécessite d'avoir des attributs dans les trois zones.

```
#include <iostream>

#include <string>

using namespace std ;

class Personne {

public :

 string nom ;
```

```
 string prenom ;
 int age ;
 char sexe ;
 string lieuDeNaissance ;
 string numeroTelephone ;
 // Fonctionnalités primaires
 void manger() { /* … */ }
 void afficher() { /* … */ }
 void dormir() { /* … */ }
 void toString() { /* … */ }
} ;
class Etudiant : public Personne {
private :
 string universite ;
 string faculte ;
 string filiere ;
 int anneeEtude ;
 string numeroMatricule ;
string adresseEmail ;
Public :
 // Fonctionnalités spécifiques à un étudiant
 void etudier() { /* … */ }
 void seFaireEvaluer() { /* … */ }
 void seDistraire() { /* … */ }
} ;
class Enseignant : public Personne {
```

```
protected :
 string specialites ;
 string matieresEnseignees ;
 string emploiDuTemps ;
public :
 // Fonctionnalités spécifiques à un enseignant
 void enseigner() { /* … */ }
 void conseiller() { /* … */ }
 void orienter() { /* … */ }
 void evaluerTravaux() { /* … */ }
} ;
class Administratif : public Personne {
private :
 string qualifications ;
 string tachesARealiser ;
 string poste ;
 string emploiDuTemps ;
public :
 // Fonctionnalités spécifiques à un administratif
 void conseiller() { /* … */ }
 void realiserTaches() { /* … */ }
} ;
class Technicien : public Personne {
private :
 string qualifications ;
 string emploiDuTemps ;
```

```
public :
 // Fonctionnalités spécifiques à un technicien
 void realiserTaches() { /* … */ }
} ;
Int main() {
 etudiant etudiant ;
etudiant.nom = ‘’AGBONNON’’ ;
 etudiant.prenom = ‘’Toto’’ ;
 etudiant.age = 20 ;
 // Tester l’accessibilité des attributs de chaque zone
 etudiant.universite = « Université XYZ » ; // Erreur de compilation, attribut privé
 etudiant.specialites = « Informatique » ; // Erreur de compilation, attribut protégé
 etudiant.qualifications = « BTS » ; // Erreur de compilation, attribut privé
 Return 0 ;
}
```

Ce code montre l’utilisation des zones public, protected et private pour les attributs, ainsi que l’héritage entre les classes. Les erreurs de compilation indiquent que l’accès aux attributs est conforme à leur niveau de visibilité défini dans les classes parentes.

Il est essentiel de retenir que les attributs privés ne sont pas accessibles directement depuis le code principal (`main`), tandis que les attributs protégés le sont uniquement dans la classe dérivée et ses sous-classes. Les attributs publics sont accessibles partout.

5.) Modifier les attributs d’un objet peut être utile pour ajuster son état interne, répondre à des changements de conditions ou personnaliser son comportement en fonction des besoins du programme. Cela offre une flexibilité dans la manipulation des objets et permet d’adapter dynamiquement leur fonctionnement.

On appelle couramment une méthode permettant de modifier les attributs d’un objet une « méthode mutatrice » ou "méthode setter ». Elle est spécialement conçue pour mettre à jour les valeurs des attributs de l’objet.

6.) Une méthode qui permet de récupérer les attributs d'un objet est souvent appelée une "méthode d'accès" ou "getter". Elle est utilisée pour obtenir la valeur d'un attribut spécifique d'un objet dans la programmation orientée objet.

7.) Modifications du programme

```
#include <iostream>
#include <string>
using namespace std ;
class Personne {
private :
 string nom ;
 string prenom ;
 int age ;
 string sexe ;
 string lieuDeNaissance ;
 string numeroTelephone ;
public :
 // Setters
 void setNom(const string& nouveauNom) {
 nom = nouveauNom ;
 }
 void setPrenom(const string& nouveauPrenom) {
 prenom = nouveauPrenom ;
 }
 void setAge(int nouvelAge) {
```

```
If (nouvelAge >= 0) {
age = nouvelAge ;
}
}
void setSexe(const string nouveauSexe) {
sexe = nouveauSexe ;
}
void setLieuDeNaissance(const string& nouveauLieu) {
lieuDeNaissance = nouveauLieu ;
}
void setNumeroTelephone(const string& nouveauNumero) {
numeroTelephone = nouveauNumero ;
}
// Getters
string getNom() const {
return nom ;
}
string getPrenom() const {
return prenom ;
}
int getAge() const {
return age ;
}
string getSexe() const {
return sexe ;
}
```

```
 string getLieuDeNaissance() const {
 return lieuDeNaissance ;
 }
 string getNumeroTelephone() const {
 return numeroTelephone ;
 }
 // Fonctions supplémentaires
 void manger() { /* … */ }
 void afficher() { /* … */ }
 void dormir() { /* … */ }
 void toString() { /* … */ }
} ;
int main() {
 personne personne ;
 personne.setNom(‘’ AGBONNON ‘’) ;
 personne.setPrenom(‘’Toto’’) ;
 personne.setAge(20) ;
 // Test des getters
 cout << ‘’ Nom : ‘’  << personne.getNom() << endl ;
 cout << ‘’ Prénom : ‘’ << personne.getPrenom() << endl ;
 cout << ‘’ Âge :’’ << personne.getAge() << endl ;
 return 0 ;
}
```

8.) Pour tester le bon fonctionnement de la classe `Personne`, on peut instancier un objet de cette classe et utiliser les méthodes définies, comme suit :

```
int main() {
```

```
// Instanciation d'un objet de la classe Personne
Personne personne ;
// Utilisation des setters
personne.setNom(‘’ AGBONNON ‘’) ;
personne.setPrenom(‘’ Toto ‘’) ;
personne.setAge(20) ;
personne.setSexe(‘M’) ;
personne.setLieuDeNaissance(‘’ Parakou ‘’) ;
personne.setNumeroTelephone(‘’ 12345678 ‘’) ;
// Utilisation des getters
cout << ‘’ Nom : ‘’ << personne.getNom() << endl ;
cout << ‘’ Prénom : ‘’ << personne.getPrenom() << endl ;
cout << ‘’ Âge : ‘’ << personne.getAge() << endl ;
cout << ‘’ Sexe : ‘’ << personne.getSexe() << endl ;
cout <<’’ Lieu de naissance : ‘’ << personne.getLieuDeNaissance() << endl ;
cout << ‘’ Numéro de téléphone : ‘’ << personne.getNumeroTelephone() <<
endl ;
// Utilisation des autres méthodes
personne.manger() ;
personne.afficher() ;
personne.dormir() ;
personne.toString() ;
return 0 ;
}
```

4.1 Héritage simple

1. Création de la classe Etudiant :

```
class Etudiant : public  Personne {
public:
   // Membres spécifiques à la classe Etudiant
};
```

2. Instanciation d'un Etudiant par une Personne :

```
Personne personne;
Etudiant etudiant = personne;
 // Erreur d'instanciation
```

Source de l'erreur :L'erreur provient de l'ambiguïté lors de l'instanciation d'un Etudiant par une Personne en raison de l'héritage simple.

* Instanciation d'une Personne par un Etudiant :

```
Etudiant etudiant;
Personne personne = etudiant;
 // Pas d'erreur
```

Résultat : L'instanciation est possible car Etudiant hérite de Personne.

3)

Ajout d'étudiants :

```
Etudiant etudiant1(/* initialiser les attributs */);
Etudiant etudiant2(/* initialiser les attributs */);
// Ajoutez d'autres étudiants si nécessaire.
```

Affichage des étudiants :

```
   // Créez un vecteur pour stocker les étudiants
vector<Etudiant> listeEtudiants;
// Ajoutez les étudiants au vecteur
listeEtudiants.push_back(etudiant1);
```

```
  listeEtudiants.push_back(etudiant2);
  // Ajoutez d'autres étudiants si nécessaire.
  // Affichez les détails de chaque étudiant
  for (const auto& etudiant : listeEtudiants) {
     etudiant.afficher(); //
  }
```

. Méthode d'affichage (`afficher()`) :

```
  // Exemple avec getters dans la classe Etudiant
  void afficherEtudiant(const Etudiant& etudiant) {
     cout << "Nom: " << etudiant.getNom() << ", Prénom: " <<
etudiant.getPrenom() << ", Age: " << etudiant.getAge() << endl;
  }
```

4. Qu'est-ce qui fait marcher une partie des instructions des questions 1, 2 et 3 :

La déclaration et l'instanciation d'une Personne par un étudiant fonctionnent car Etudiant hérite de Personne.

5. Finalisation de la construction de la classe Etudiant :

```
  class Etudiant : public Personne {
  public:
     // Membres spécifiques à la classe Etudiant
     string universite;
     string faculte;
     string filiere;
     int anneeEtude;
     string numeroMatricule;
     string adresseEmail;
     // Autres membres
```

};

6. Construction systématique de la partie Personne d'un Etudiant

Utiliser le constructeur de la classe Personne dans le constructeur d'Etudiant en utilisant la liste d'initialisation.

```
Etudiant::Etudiant(...) : Personne(...) {
    // Initialisation des membres spécifiques à Etudiant
}
```

7. Commenter la méthode toString dans la classe Etudiant et afficher un Etudiant :

```
// Commenter la méthode toString
// ...
// Afficher un Etudiant en utilisant la méthode
Etudiant etudiant;
etudiant.afficher();
```

8. Pourquoi ça marche quand même ?

Cela fonctionne car la classe Etudiant hérite de la méthode toString de la classe Personne.

9. Compter le nombre d'instances de chaque classe :

Utilisez des variables statiques pour compter le nombre d'instances dans chaque classe.

10. Adapter constructeurs et destructeurs pour afficher les nombres d'instances :

Utilisez des messages d'affichage dans les constructeurs et destructeurs pour suivre le nombre d'instances créées et détruites.

11. Que se passe-t-il à la création et destruction d'un Etudiant ?

Les messages d'affichage dans les constructeurs et destructeurs montrent le nombre d'instances créées et détruites.

12. Finalisation des classes Enseignant, Administratif et Technicien :

```
class Enseignant : public Personne {
public:
    // Membres spécifiques à la classe Enseignant
    // ...
};
class Administratif : public Personne {
public:
    // Membres spécifiques à la classe Administratif
    // ...
};
class Technicien : public Personne {
public:
    // Membres spécifiques à la classe Technicien
    // ...
} ;
```

4.2 Héritage multiple

1. Construction de la classe des Moniteurs :

```
class Moniteur : public Etudiant, public Enseignant {
public:
    // Membres spécifiques à la classe Moniteur
    // ...
};
```

2. Problème lors de l'instanciation d'un Moniteur :

L'ambiguïté des membres communs entre Etudiant et Enseignant peut provoquer des erreurs lors de l'instanciation d'un Moniteur.

3. Utiliser des classes virtuelles pour régler le problème :

Déclarez des classes virtuelles dans Etudiant et Enseignant pour résoudre l'ambiguïté. Testez la solution en instanciant un Moniteur.

```
class Moniteur : public virtual Etudiant, public virtual Enseignant {
public:
    // Membres spécifiques à la classe Moniteur
    // ...
};
```

5.1 Héritage et accès aux membres d'une classe mère

Dans cet exercice , nous allons essayer de regrouper touts les questions en un seul code afin de faciliter la lecture…….

```
#include <iostream>

// Classe Mere
class Mere {
public:
    int attrPubMere;
private:
    int attrPrivMere;
protected:
    int attrProtMere;

public:
    Mere() {
        cout << "Utilisation du constructeur par défaut de Mère." << endl;
    }

    Mere(int a, int b, int c) : attrPubMere(a), attrPrivMere(b), attrProtMere(c)
{
        cout << "Utilisation du constructeur de trois arguments de Mère." <<
endl;
    }

    ~Mere() {
        cout << "Destruction d'une instance de Mère." << endl;
```

```
    }

    void afficher() {
        cout << "Attributs de Mère: " << attrPubMere << ", " << attrPrivMere <<
", " << attrProtMere << endl;
    }

    void checkPolyMorphisme() {
        cout << "Fonction checkPolyMorphisme 'checkant' de la classe Mere."
<< endl;
    }
};

// Classes FilleX de Mere
class FillePub : public Mere {
public:
    FillePub() {
        cout << "Utilisation du constructeur par défaut de FillePub." << endl;
    }

    FillePub(int a, int b, int c, int d, int e, int f) : Mere(a, b, c), attrPubFilleX(d),
attrPrivFilleX(e), attrProtFilleX(f) {
        cout << "Utilisation du constructeur de six arguments de FillePub." <<
endl;
    }

    ~FillePub() {
        cout << "Destruction d'une instance de FillePub." << endl;
    }

    void afficher() {
        Mere::afficher();
        cout << "Attributs de FillePub: " << attrPubFilleX << ", " <<
attrPrivFilleX << ", " << attrProtFilleX << endl;
    }

    void modifierAttributs() {
        attrPubMere = 11;   // Accès public de la classe de base
        // attrPrivMere = 22;  // Erreur - Accès privé de la classe de base
(commenté pour éviter l'erreur)
        attrProtMere = 33;  // Accès protégé de la classe de base
        attrPubFilleX = 44;
```

```
        attrPrivFilleX = 55;
        attrProtFilleX = 66;
    }

    void checkPolyMorphisme() {
        cout << "Fonction checkPolyMorphisme 'checkant' de la classe
FillePub." << endl;
    }

public:
    int attrPubFilleX;
private:
    int attrPrivFilleX;
protected:
    int attrProtFilleX;
};

// Fonction principale

int main() {

    // Création d'instances
    Mere mere;
    mere.afficher();
    mere.checkPolyMorphisme();

    FillePub fillePub;
    fillePub.afficher();
    fillePub.modifierAttributs();
    fillePub.afficher();
    fillePub.checkPolyMorphisme();


return 0;

}
```

3-) Messages d'erreurs :

•Il n'y a pas d'erreur syntaxique dans le code fourni.

4-) Compter les instances de Mere créées et détruites :

Pour compter les instances de Mere créées et détruites, on doit ajouter des compteurs statiques dans la classe Mere et incrémenter/décrémenter ces compteurs dans le constructeur et le destructeur respectivement.

5-) Instances créées de Mere et FillePub :

Si on crée une instance de chaque classe, on aura 7 instances au total (1 Mere, 3 FillePub, 1 FillePriv, 1 FillePubPub, 1 FilleProtProt). Cela inclut également les instances détruites.

6-) Appeler le constructeur de Mere avec trois arguments :

NOUS devons appeler le constructeur de Mere avec trois arguments dans le constructeur de chaque classe FilleX ayant six arguments. Cela doit être fait dans la liste d'initialisation du constructeur de la classe FilleX .

7-) Méthode checkPolyMorphisme :

Ajoutons la méthode checkPolyMorphisme dans chaque classe avec le bon message. Appelons cette méthode dans chaque classe pour voir le polymorphisme en action.

```
public:

   void checkPolyMorphisme() {

      cout << "Fonction checkPolyMorphisme 'checkant' de la classe Mere." << endl;

   }
```

Ensuite ,on doit appeler cette méthode dans chaque classe :

(a) Mere : checkPolyMorphisme();

(b) FillePriv : checkPolyMorphisme();

(c) FillePubPub : checkPolyMorphisme();

(d) FilleProtProt : checkPolyMorphisme();

5-2 Héritage et propagation de l'héritage

Pour mettre en exergue les mécanismes d'héritage avec différents types d'accès (public, protected, private), nous pouvons suivre ces étapes en utilisant les classes `Mere`, `FillePub`, `FillePriv`, et `FilleProt` définies dans l'exemple :

1. Héritage avec accès public :

```
FillePub fillePub;
fillePub.attrPubMere = 1;  // Accès public
fillePub.afficher();       // Affiche les attributs de Mere
```

2. Héritage avec accès protected :

```
FilleProt filleProt;
// filleProt.attrPubMere = 1;  // Erreur - Accès public transformé en protected
filleProt.afficher();          // Affiche les attributs de Mere (en protected)
```

3. Héritage avec accès private :

```
FillePriv fillePriv;
// fillePriv.attrPubMere = 1;  // Erreur - Accès public transformé en private
// fillePriv.attrProtMere = 2;  // Erreur - Accès protected transformé en private
// fillePriv.afficher();        // Erreur - Impossible d'accéder aux membres de Mere
```

Ces exemples illustrent comment différents types d'accès lors de l'héritage affectent la visibilité des membres de la classe de base (`Mere` dans ce cas) dans les classes dérivées (`FillePub`, `FillePriv`, `FilleProt`).

VOICI LE PROGRAMME FINAL :

```cpp
#include <iostream>
#include <string>
using namespace std;

class Personne {
private:
    string nom;
    string prenom;
    int age;
    string sexe;
    string lieuDeNaissance;
```

```
    string numeroTelephone;

public:
    // Setters
    void setNom(const string& nouveauNom) {
        nom = nouveauNom;
    }

    void setPrenom(const string& nouveauPrenom) {
        prenom = nouveauPrenom;
    }

    void setAge(int nouvelAge) {
        if (nouvelAge >= 0) {
            age = nouvelAge;
        }
    }

    void setSexe(const string& nouveauSexe) {
        sexe = nouveauSexe;
    }

    void setLieuDeNaissance(const string& nouveauLieu) {
        lieuDeNaissance = nouveauLieu;
    }

    void setNumeroTelephone(const string& nouveauNumero) {
        numeroTelephone = nouveauNumero;
    }

    // Getters
    string getNom() const {
        return nom;
    }

    string getPrenom() const {
        return prenom;
    }

    int getAge() const {
        return age;
    }

    string getSexe() const {
        return sexe;
    }

    string getLieuDeNaissance() const {
        return lieuDeNaissance;
    }

    string getNumeroTelephone() const {
        return numeroTelephone;
    }

    // Fonctions supplémentaires
    void manger() {
        cout << nom << " " << prenom << " mange." << endl;
    }

    void afficher() {
```

```
        cout << nom << " " << prenom << " affiche quelque chose." << endl;
    }

    void dormir() {
        cout << nom << " " << prenom << " dort." << endl;
    }

    void toString() {
        cout << "Nom: " << nom << ", Prenom: " << prenom << ", Age: " <<
age
             << ", Sexe: " << sexe << ", Lieu de naissance: " <<
lieuDeNaissance
             << ", Numero de telephone: " << numeroTelephone << endl;
    }
};

class Etudiant : public virtual Personne {
private:
    string universite;
    string faculte;
    string filiere;
    int anneeEtude;
    string numeroMatricule;
    string adresseEmail;

public:
    // Setters
    void setUniversite(const string& nouvelleUniversite) {
        universite = nouvelleUniversite;
    }

    void setFaculte(const string& nouvelleFaculte) {
        faculte = nouvelleFaculte;
    }

    void setFiliere(const string& nouvelleFiliere) {
        filiere = nouvelleFiliere;
    }

    void setAnneeEtude(int nouvelleAnnee) {
        if (nouvelleAnnee >= 1) {
            anneeEtude = nouvelleAnnee;
        }
    }

    void setNumeroMatricule(const string& nouveauNumeroMatricule) {
        numeroMatricule = nouveauNumeroMatricule;
    }

    // Getters (inchangés)

    // Fonctions spécifiques à un étudiant
    void etudier() {
        cout << getNom() << " " << getPrenom() << " etudie." << endl;
    }

    void seFaireEvaluer() {
        cout << getNom() << " " << getPrenom() << " se fait evaluer." <<
endl;
    }
```

```
    void seDistraire() {
        cout << getNom() << " " << getPrenom() << " se distrait." << endl;
    }
};

class Enseignant : public virtual Personne {
private:
    string specialites;
    string matieresEnseignees;
    string emploiDuTemps;

public:
    // Setters
    void setSpecialites(const string& nouvellesSpecialites) {
        specialites = nouvellesSpecialites;
    }

    void setMatieresEnseignees(const string& nouvellesMatieres) {
        matieresEnseignees = nouvellesMatieres;
    }

    void setEmploiDuTemps(const string& nouvelEmploiDuTemps) {
        emploiDuTemps = nouvelEmploiDuTemps;
    }

    // Getters (inchangés)

    // Fonctions spécifiques à un enseignant
    void enseigner() {
        cout << getNom() << " " << getPrenom() << " enseigne." << endl;
    }

    void conseiller() {
        cout << getNom() << " " << getPrenom() << " conseille." << endl;
    }

    void orienter() {
        cout << getNom() << " " << getPrenom() << " oriente." << endl;
    }

    void evaluerTravaux() {
        cout << getNom() << " " << getPrenom() << " evalue les travaux." <<
endl;
    }
};

class Administratif : public Personne {
private:
    string qualifications;
    string taches;
    string poste;
    string emploiDuTemps;

public:
    // Setters
    void setQualifications(const string& nouvellesQualifications) {
        qualifications = nouvellesQualifications;
    }

    void setTaches(const string& nouvellesTaches) {
        taches = nouvellesTaches;
```

```
    }

    void setPoste(const string& nouveauPoste) {
        poste = nouveauPoste;
    }

    void setEmploiDuTemps(const string& nouvelEmploiDuTemps) {
        emploiDuTemps = nouvelEmploiDuTemps;
    }

    // Getters (inchangés)

    // Fonctions spécifiques à un administratif
    void conseiller() {
        cout << getNom() << " " << getPrenom() << " conseille." << endl;
    }

    void realiserTaches() {
        cout << getNom() << " " << getPrenom() << " realise ses taches." <<
endl;
    }
};

class Technicien : public Personne {
private:
    string qualifications;
    string emploiDuTemps;

public:
    // Setters
    void setQualifications(const string& nouvellesQualifications) {
        qualifications = nouvellesQualifications;
    }

    void setEmploiDuTemps(const string& nouvelEmploiDuTemps) {
        emploiDuTemps = nouvelEmploiDuTemps;
    }

    // Getters (inchangés)

    // Fonctions spécifiques à un technicien
    void realiserTaches() {
        cout << getNom() << " " << getPrenom() << " realise ses taches
techniques." << endl;
    }
};
// Classe Moniteur dérivée de Etudiant et Enseignant (Héritage multiple
avec classes virtuelles)
class Moniteur : public virtual Etudiant, public virtual Enseignant {
public:
    // Les méthodes et attributs spécifiques à la classe Moniteur
};
// Classe Mere
class Mere {
public:
    int attrPubMere;
private:
    int attrPrivMere;
protected:
    int attrProtMere;
```

```
public:
    Mere() {
        cout << "Utilisation du constructeur par defaut de Mere." << endl;
    }

    Mere(int a, int b, int c) : attrPubMere(a), attrPrivMere(b),
attrProtMere(c) {
        cout << "Utilisation du constructeur de trois arguments de Mere."
<< endl;
    }

    ~Mere() {
        cout << "Destruction d une instance de Mere." << endl;
    }

    void afficher() {
        cout << "Attributs de Mere: " << attrPubMere << ", " <<
attrPrivMere << ", " << attrProtMere << endl;
    }

    void checkPolyMorphisme() {
        cout << "Fonction checkPolyMorphisme 'checkant' de la classe Mere."
<< endl;
    }
};

// Classes FilleX de Mere
class FillePub : public Mere {
public:
    FillePub() {
        cout << "Utilisation du constructeur par defaut de FillePub." <<
endl;
    }

    FillePub(int a, int b, int c, int d, int e, int f) : Mere(a, b, c),
attrPubFilleX(d), attrPrivFilleX(e), attrProtFilleX(f) {
        cout << "Utilisation du constructeur de six arguments de FillePub."
<< endl;
    }

    ~FillePub() {
        cout << "Destruction d une instance de FillePub." << endl;
    }

    void afficher() {
        Mere::afficher();
        cout << "Attributs de FillePub: " << attrPubFilleX << ", " <<
attrPrivFilleX << ", " << attrProtFilleX << endl;
    }

    void modifierAttributs() {
        attrPubMere = 11;   // Accès public de la classe de base
        // attrPrivMere = 22;  // Erreur - Accès privé de la classe de base
(commenté pour éviter l'erreur)
        attrProtMere = 33;  // Accès protégé de la classe de base
        attrPubFilleX = 44;
        attrPrivFilleX = 55;
        attrProtFilleX = 66;
    }

    void checkPolyMorphisme() {
```

```
        cout << "Fonction checkPolyMorphisme 'checkant' de la classe
FillePub." << endl;
    }

public:
    int attrPubFilleX;
private:
    int attrPrivFilleX;
protected:
    int attrProtFilleX;
};

int main() {

    Moniteur moniteur;
    moniteur.setNom("AGBONNON");
    moniteur.setPrenom("Toto");
    moniteur.setAge(25);
    moniteur.setSexe("M");
    moniteur.setLieuDeNaissance("Parakou");
    moniteur.setNumeroTelephone("12345678");
    moniteur.setUniversite("UP");
    moniteur.setFaculte("IUT");
    moniteur.setFiliere("Informatique");
    moniteur.setAnneeEtude(2);
    moniteur.setNumeroMatricule("1557");

    moniteur.setSpecialites("Informatique");
    moniteur.setMatieresEnseignees("Programmation");
    moniteur.setEmploiDuTemps("8h à 14H");

    moniteur.toString();
    moniteur.etudier();
    moniteur.seFaireEvaluer();
    moniteur.seDistraire();
    moniteur.enseigner();
    moniteur.conseiller();
    moniteur.orienter();
    moniteur.evaluerTravaux();


    Administratif administratif;
    administratif.setNom("BEHANZIN");
    administratif.setPrenom("Bere");
    administratif.setAge(30);
    administratif.setSexe("F");
    administratif.setLieuDeNaissance("Abidjan");
    administratif.setNumeroTelephone("23456789");
    administratif.setQualifications("Gestion");
    administratif.setTaches("Taches Administratives");
    administratif.setPoste("Poste Z");
    administratif.setEmploiDuTemps("8h à 12h");

    administratif.toString();
    administratif.conseiller();
    administratif.realiserTaches();

    Technicien technicien;
    technicien.setNom("GBOGBO");
    technicien.setPrenom("Luc");
    technicien.setAge(40);
```

```
    technicien.setSexe("M");
    technicien.setLieuDeNaissance("Natittingou");
    technicien.setNumeroTelephone("43246534");
    technicien.setQualifications("Maintenance");
    technicien.setEmploiDuTemps("Non defini");

    technicien.toString();
    technicien.realiserTaches();

    Mere mere;
        mere.afficher();
        mere.checkPolyMorphisme();

        FillePub fillePub;
        fillePub.afficher();
        fillePub.modifierAttributs();
        fillePub.afficher();
        fillePub.checkPolyMorphisme();



    return 0;
}
```