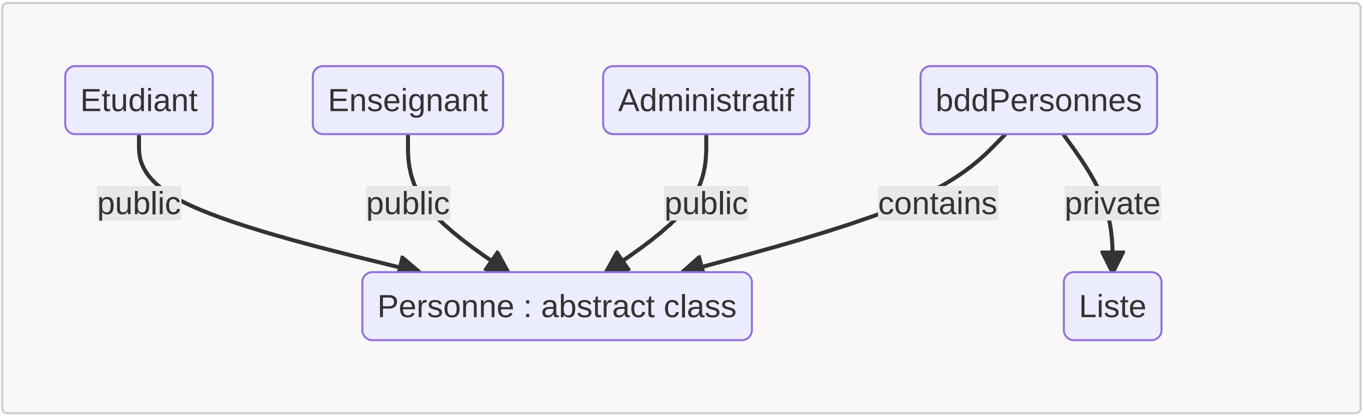
TP3 - Database of peoples

Introduction

For this third practical session, we have to build a small database in C++ using the principles of inheritance. We could choose the subject we wanted but I choose the subject given : a database of the UTBM. In the UTBM, we have some loads of people. This people are in fact, either students, professors or administrative people. All these functions will derived from the `Personne` class. We also have the database which will inherit privately from a list and this list will contain pointers on people. Our list do not contain Personne because inheritance in C++ requires to use pointe on the abstract class.



Abstract class : Personne

Firstly I designed, thanks to the files that were given, the abstract class `Personne`. This class is called an abstract class because she has at least **one** `abstract method`.

Attributes

The class have some basic attributes :

```
class Personne {
        protected:
                string _nom ;
                string _prenom ;
                string _qualite;
                Liste<string> _experiences;
        public:
            [...]
```

These attributes are the things that everyone has when he's in an university :

- A lastname (`nom`)
- A firstname (`prenom`)
- His/Her task (`qualité`)
- His/Her experiences (`expériences`) : A list of experiences that the people have been through.

Of course, these attributes ar not the only one because each catgory of people has its own specificities.

Essential methods

For this class, two methods are strictly necessary :

- the constructor by default : Without him, it impossible to generate a correct instance.
- the destructor : it has to be `virtual` in order to be usable by the derived class.

```
Personne(string nom, string prenom, string qualite) : _nom(nom),
_prenom(prenom), _qualite(qualite) {}
virtual ~Personne() {}
```

The classic methods

As a first step, I designed some classic functions for the `Personne` class :

```
string getNom() { return _nom; }
void experience();
void ajouterExperience(string exp);
```

These methods are common to all the classes that will inherit from `Personne`.

Implementation file:

```
void Personne::experience() {
    Liste<string>::ListeIterateur it(_experiences);
    cout << "Expériences : " << endl;
    while(it.hasNext())
        cout << "- " << it.get() << endl;
}

void Personne::ajouterExperience(string exp) {
    _experiences.ajouter(exp, 0);
}
```

These functions are here to display and add something to the experiences of someone.

The virtual methods

We build virtual function in order to use th mechnaism of polymorphism.

Header file :

```
virtual void affiche();
virtual void qualite();
```

Implementation file :

```
void Personne::affiche() {
    cout << "Mon nom est " << _nom <<" et mon prenom " << _prenom << " ";
}

void Personne::qualite() {
    cout << "Je suis " << _qualite;
}
```

These two functions are used to display the characteristics of someone. But every type of people have different characteristics, so we just start the sentences.

The abstract method

This class is an abstract class because it has an abstract method.

```
virtual void profession() = 0;
```

This class has no definition in the `Personne` class and every class that is deived from `Personne` should implement the method `profession`.

The insertion operator

It is not possible to define the insertion operator (`operator<<`) in a class and hope it will be derived properly. For this reason, we cannot make this function `virtual`. Instead, we use a small trck to have a similar behaviour :

Header file :

```
protected:
        [...];
        virtual void afficherPersonne(ostream&);
public:
        [...];
        friend ostream& operator<<(ostream&, Personne&);
```

First, we define a virtual protected method that will be callable by all the derived classes but not by a client file. Then, we define the insertion operator for the class `Personne` and all the derived classes. The `operator<<` will call the `affichePersonne` method of the current class.

Implementation file :

```
void Personne::afficherPersonne(ostream& os) {
    os << "Nom : " << _nom << endl;
    os << "Prénom : " << _prenom << endl;
    os << "Qualité : " << _qualite << endl;
}

ostream& operator<<(ostream& os, Personne& pers) {
    pers.afficherPersonne(os);
    return os;
}
```

Students, Teacher, Administrative classes

In this part, we will discuss about the implementation of the virtual functions in each class.

Added parameters

Student class

In the student class, we just added the branch and the sector (ih he is in the end of his studies) of a student at UTBM :

```
class Etudiant : public Personne {
    private :
        string _branche;
        string _filiere;
                [...]
```

Teacher class

In the teacher class, we added the type of formation where he teaches, his laboatory (if he is a professor and a researcher) and his office.

```
class Enseignant : public Personne {
    private :
        string _type_formation;
        string _laboratoire;
                string _bureau;
```

Administrative class

For the administrative classes, we added the attributes defining his function (what is his job at UTBM) and also his office.

```
class Administratif : public Personne {
    private :
        string _fonction;
                string _bureau;
```

The constructors

The constructors are slightly different for each derived class :

Student class

```
Etudiant(string prenom,
        string nom,
        string branche,
        string filiere) : Personne(nom, prenom, "étudiant(e)"),
_branche(branche), _filiere(filiere) {}

Etudiant(string prenom,
        string nom,
        string branche) : Personne(nom, prenom, "étudiant(e)"),
_branche(branche), _filiere() {}
```

We have two constructors :

- One is used for the students that reach the end of their curriculum university.
- The second is used for the students that haven't reached their last study year.

Teacher class

```
Enseignant(string prenom,
        string nom,
        string type_formation,
        string laboratoire,
        string bureau) : Personne(nom, prenom, "enseignant(e)-chercheur"),
_type_formation(type_formation), _laboratoire(laboratoire), _bureau(bureau)
{}

Enseignant(string nom,
        string prenom,
        string type_formation,
        string bureau) : Personne(nom, prenom, "professeur(e) agrégé(e)"),
_type_formation(type_formation), _laboratoire(), _bureau(bureau) {}
```

We also have two constructors :

- One is used for the aggregated teachers that do not make researches.

- The second is used for the professors who are also researchers.

Administrative class

```
Administratif(string prenom,
        string nom,
        string fonction,
        string bureau) : Personne(nom, prenom, "personnel administratif"),
_fonction(fonction), _bureau(bureau) {}
```

For this one, we have only one constructor, he works in the same way as the other constructors with no particular specificity.

Creation of objects in the client file

```
Personne* etd1 = new Etudiant("Jean", "Pierre", "Info", "CS12");
Personne* etd2 = new Etudiant("Alexandre","Viala","Info");
Personne* etd3 = new Etudiant("Amelie ", "Rodriguez", "Info", "CS13");

Personne* ensg1 = new Enseignant("Lewis", "Hamilton", "FISE" , "CIAD Lab" ,
"M405");
Personne* ensg2 = new Enseignant("Alexis", "Flech", "TC", "P101");

Personne* admin1 = new Administratif("Elanor", "Courtney", "IT-Support",
"H023");
```

We can see that we use the polymorphism's mechanism by creating pointers on `Personne` to take advantage of a dynamic definition. We also use the different constructors to have a modularity even inside each class.

The `affiche` and the `qualite` methods

The `qualite` method is only redifined in order to add a simple `endl` escape sequence to the result of the function :

```
void Etudiant::affiche(void) {
    Personne::affiche();
    cout << "et je suis étudiant" << endl;
}

void Enseignant::qualite() {
    Personne::qualite();
    cout << endl;
}

void Administratif::qualite() {
    Personne::qualite();
    cout << endl;
}
```

The `affiche` method will re-used the content of the inital function for each class and then add its own specificity.

In the original class, it wass defined in this way :

```
void Personne::affiche() {
    cout << "Mon nom est " << _nom <<" et mon prenom " << _prenom << " ";
}
```

In the Student class

```
void Etudiant::affiche(void) {
    Personne::affiche();
    cout << "et je suis étudiant" << endl;
}
```

In the Teacher class

```
void Enseignant::affiche(){
    Personne::affiche();
    cout << "et je suis enseignant" << endl;
}
```

In the Administrative class

```
void Administratif::affiche(){
    Personne::affiche();
    cout << "et je suis un personnel administratif" << endl;
}
```

Demonstration in the client file

We write this code to test these methods :

```
/* Test of the "affiche()" method */
etd1->affiche();
etd2->affiche();
etd3->affiche();
```

```
ensg1->affiche();
ensg2->affiche();
admin1->affiche();

cout << endl;

/* Test of the "qualite()" method */
etd1->qualite();
etd2->qualite();
etd3->qualite();
ensg1->qualite();
ensg2->qualite();
admin1->qualite();
```

Output :

```
Mon nom est Pierre et mon prenom Jean et je suis étudiant
Mon nom est Viala et mon prenom Alexandre et je suis étudiant
Mon nom est Rodriguez et mon prenom Amelie  et je suis étudiant
Mon nom est Hamilton et mon prenom Lewis et je suis enseignant
Mon nom est Alexis et mon prenom Flech et je suis enseignant
Mon nom est Courtney et mon prenom Elanor et je suis un personnel
administratif

Je suis étudiant(e)
Je suis étudiant(e)
Je suis étudiant(e)
Je suis enseignant(e)-chercheur
Je suis professeur(e) agrégé(e)
Je suis personnel administratif
```

The `profession` method

In the original class, this function was defined as an abstract method. Therefore, it is mandatory to add a definition to it in every derived classes.

In the Student class

```
void Etudiant::profession(){
    Personne::qualite();
    cout << " et je suis en " << _branche;
    if(!_filiere.empty())
        cout << " en filière " << _filiere;
    cout << endl;
}
```

Here we take 2 cases in consideration :

- The student is in his last study year, so he is in a particular sector and we can print it.
- The student is not yet in his last year, so he does not have a particular sector.

In the Teacher class

```
void Enseignant::profession(){
    Personne::qualite();
    if (_laboratoire.empty())
        cout << " en " << _type_formation << " et mon bureau est le " <<
_bureau << endl;
    else
        cout << " en " << _type_formation << " au laboratoire " <<
_laboratoire << " et mon bureau est le " << _bureau << endl;
}
```

We also take 2 cases in consideration :

- The teacher is an aggregated professor so he does not have a laboratory.
- he is a professor and a researcher so he has a laoratory.

In the Administrative class

```
void Administratif::profession(){
    Personne::qualite();
    cout << " en tant que " << _fonction <<  " et mon bureau est le " <<
_bureau << endl;
}
```

In this function, the functionning is really straightforward.

Demonstration in the client file

```
etd1->profession();
etd2->profession();
etd3->profession();
ensg1->profession();
ensg2->profession();
admin1->profession();
```

Output :

```
Je suis étudiant(e) et je suis en Info en filière CS12
Je suis étudiant(e) et je suis en Info
Je suis étudiant(e) et je suis en Info en filière CS13
Je suis enseignant(e)-chercheur en FISE au laboratoire CIAD Lab et mon
```

```
bureau est le M405
Je suis professeur(e) agrégé(e) en TC et mon bureau est le P101
Je suis personnel administratif en tant que IT-Support et mon bureau est le
H023
```

The `experience` method

According to what I said earlier in this report, I did not redefine the `experience` and `ajouterExperience` methods.

Demonstration in client file

```
Personne* etd2 = new Etudiant("Alexandre","Viala","Info");
etd2->ajouterExperience("ST10 chez Exxelia");
etd2->ajouterExperience("ST40");

Personne* ensg2 = new Enseignant("Alexis", "Flech", "TC", "P101");
ensg2->ajouterExperience("Classe préparatoire");
ensg2->ajouterExperience("Enseignant de mathématiques");

/* Test of the "experience()" method */
    etd1->experience();
    etd2->experience();
    etd3->experience();
    ensg1->experience();
    ensg2->experience();
    admin1->experience();
```

Output :

```
Expériences :
Expériences :
- ST40
- ST10 chez Exxelia
Expériences :
Expériences :
Expériences :
- Enseignant de mathématiques
- Classe préparatoire
Expériences :
```

The `afficherPersonne` method

It is the most interesting method : it is the virtual method called by the insertion operator.

In the Person class

```cpp
void Personne::afficherPersonne(ostream& os) {
    os << "Nom : " << _nom << endl;
    os << "Prénom : " << _prenom << endl;
    os << "Qualité : " << _qualite << endl;
}
```

We display The last name, the first name and the profession of the person. Then we display the particularities of each derived classes.

In the Student class

```cpp
void Etudiant::afficherPersonne(ostream& os) {
    Liste<string>::ListeIterateur it(_experiences);

    Personne::afficherPersonne(os);
    os << "Branche : " << _branche << endl;
    if(!_filiere.empty())
        os << "Filière : " << _filiere << endl;

    os << "Expériences : " << endl;
    while(it.hasNext())
        os << "- " << it.get() << endl;
}
```

We make all the things made in the other functions but in a more minimalsit approach. We re-write some things that are already in other functions because we want to output all the informations in an output stream whatever it is. In this way, it is easy to output these information in a txt file by example.

In the Teacher class

```cpp
void Enseignant::afficherPersonne(ostream& os) {
    Liste<string>::ListeIterateur it(_experiences);

    Personne::afficherPersonne(os);
    os << "Type de formation : " << _type_formation << endl;
    os << "Bureau : " << _bureau << endl;
    if (!_laboratoire.empty())
        os << "Laboratoire : " << _laboratoire << endl;

    os << "Expériences : " << endl;
    while(it.hasNext())
        os << "- " << it.get() << endl;
}
```

In the Administrative class

```
void Administratif::afficherPersonne(ostream& os) {
    Liste<string>::ListeIterateur it(_experiences);

    Personne::afficherPersonne(os);
    os << "Fonction : " << _fonction << endl;

    os << "Expériences : " << endl;
    while(it.hasNext())
        os << "- " << it.get() << endl;
}
```

Demonstration in cleint file

This function is mainly used in the BDD, defined shortly after. So I will make a demonstration in the next part of this function.

Database class

Finally, to classify this data in an optimized way, we defined a Database class that will define a classic database of persons :

```
class BDD : protected Liste<Personne*> {
public:
    void ajouter(Personne* p);

    void afficheContenuBDD();
};
```

This class inherit from a list of pointers on person. We store pointers rather than static objects because we cannot define a list which contains different static types but we can easily store different types of dynamic object in one list. We inherit from list in a `protected` way because we do not want that we can access the original functions of the list.

`ajouter` method

```
void BDD::ajouter(Personne* p) {
    Liste::ajouter(p, longueur());
}
```

This method simply add something in the list object. We add this element at the end of the database to follow a chronogical order.

`afficheContenuBDD` method

```
void BDD::afficheContenuBDD() {
    // Parcourir la liste des personnes et afficher leurs informations
    Liste<Personne*>::ListeIterateur it(*this);
    Personne* cur_pers = NULL;
    int i = 0;
    cout <<
"**************************************************************************
******" << endl;
    cout << "**************** Database UTBM :
***********************************************" << endl;
    cout <<
"**************************************************************************
******" << endl << endl;
    while(it.hasNext()){
        cout << "\tEntrée " << ++i << " :" << endl << endl;
        cur_pers = it.get();
        cout << *cur_pers << endl;
    }
    cout <<
"**************************************************************************
******" << endl;
    cout << "**************** End of DATABASE
***********************************************" << endl;
    cout <<
"**************************************************************************
******" << endl << endl;
}
```

This method is the most important in this class. This method will print the content of the database. It will iterate through the database with an iterator and call the insertion operator (which call the `afficherPersonne` method) for every person in the database.

Demonstration in the client file

```
bdd.ajouter(etd1);
bdd.ajouter(etd2);
bdd.ajouter(etd3);
bdd.ajouter(ensg1);
bdd.ajouter(ensg2);
bdd.ajouter(admin1);

bdd.afficheContenuBDD();
```

Output :

```
***************************************************************************
*****
**************** Database UTBM :
```

```
***********************************************
****************************************************************************
*****

        Entrée 1 :

Nom : Pierre
Prénom : Jean
Qualité : étudiant(e)
Branche : Info
Filière : CS12
Expériences :

        Entrée 2 :

Nom : Viala
Prénom : Alexandre
Qualité : étudiant(e)
Branche : Info
Expériences :
- ST40
- ST10 chez Exxelia

        Entrée 3 :

Nom : Rodriguez
Prénom : Amelie
Qualité : étudiant(e)
Branche : Info
Filière : CS13
Expériences :

        Entrée 4 :

Nom : Hamilton
Prénom : Lewis
Qualité : enseignant(e)-chercheur
Type de formation : FISE
Bureau : M405
Laboratoire : CIAD Lab
Expériences :

        Entrée 5 :

Nom : Alexis
Prénom : Flech
Qualité : professeur(e) agrégé(e)
Type de formation : TC
Bureau : P101
Expériences :
- Enseignant de mathématiques
- Classe préparatoire

        Entrée 6 :
```

```
Nom : Courtney
Prénom : Elanor
Qualité : personnel administratif
Fonction : IT-Support
Expériences :

*****************************************************************************
*****************
**************** End of DATABASE
************************************************
*****************************************************************************
*****
```

Appendix

Personne

Header file

```
#ifndef __PERSONNE_H
#define __PERSONNE_H
#include <iostream>
#include "ListeRecursif.h"
#include <ostream>
#include <string>

using namespace std;

class Personne {
    protected:
        string _nom ;
        string _prenom ;
        string _qualite;
        Liste<string> _experiences;
        virtual void afficherPersonne(ostream&);
    public:
        Personne(string nom, string prenom, string qualite) : _nom(nom),
_prenom(prenom), _qualite(qualite) {}
        virtual ~Personne() {}
        string getNom() { return _nom; }
        void experience();
        void ajouterExperience(string exp);

        virtual void affiche();
        virtual void profession() = 0;
        virtual void qualite();


        friend ostream& operator<<(ostream&, Personne&);
};
```

```
#endif
```

Implementation file

```
#include "header/Personne.h"
#include <ostream>

void Personne::affiche() {
    cout << "Mon nom est " << _nom <<" et mon prenom " << _prenom << " ";
}

void Personne::qualite() {
    cout << "Je suis " << _qualite;
}
void Personne::ajouterExperience(string exp) {
    _experiences.ajouter(exp, 0);
}

void Personne::experience() {
    Liste<string>::ListeIterateur it(_experiences);
    cout << "Expériences : " << endl;
    while(it.hasNext())
        cout << "- " << it.get() << endl;
}

void Personne::afficherPersonne(ostream& os) {
    os << "Nom : " << _nom << endl;
    os << "Prénom : " << _prenom << endl;
    os << "Qualité : " << _qualite << endl;
}

ostream& operator<<(ostream& os, Personne& pers) {
    pers.afficherPersonne(os);
    return os;
}
```

Etudiant

Header file

```
#ifndef __ETUDIANT_H
#define __ETUDIANT_H

#include "Personne.h"
#include <ostream>

using namespace std;
```

```
class Etudiant : public Personne {
    private :
        // donnees specifiques a etudiant
        string _branche;
        string _filiere;
                virtual void afficherPersonne(ostream&);

    public :
        Etudiant(string prenom,
                 string nom,
                 string branche,
                 string filiere) : Personne(nom, prenom, "étudiant(e)"),
_branche(branche), _filiere(filiere) {}

        Etudiant(string prenom,
                 string nom,
                 string branche) : Personne(nom, prenom, "étudiant(e)"),
_branche(branche), _filiere() {}

        virtual void affiche(void);

        virtual void profession();
        virtual void qualite();
};

#endif
```

Implementation file

```
#include "header/Etudiant.h"
#include <ostream>

void Etudiant::affiche(void) {
    Personne::affiche();
    cout << "et je suis étudiant" << endl;
}

void Etudiant::profession(){
    Personne::qualite();
    cout << " et je suis en " << _branche;
    if(!_filiere.empty())
        cout << " en filière " << _filiere;
    cout << endl;
}

void Etudiant::qualite() {
    Personne::qualite();
    cout << endl;
}
```

```
void Etudiant::afficherPersonne(ostream& os) {
    Liste<string>::ListeIterateur it(_experiences);

    Personne::afficherPersonne(os);
    os << "Branche : " << _branche << endl;
    if(!_filiere.empty())
        os << "Filière : " << _filiere << endl;

    os << "Expériences : " << endl;
    while(it.hasNext())
        os << "- " << it.get() << endl;
}
```

Enseignant

Header file

```
#ifndef __ENSEIGNANT_H
#define __ENSEIGNANT_H

#include "Personne.h"

using namespace std;

class Enseignant : public Personne {
    private :
        // donnees specifiques a enseignant
        string _type_formation;
        string _laboratoire;
                string _bureau;
                virtual void afficherPersonne(ostream&);

    public :
        Enseignant(string prenom,
                  string nom,
                  string type_formation,
                  string laboratoire,
                  string bureau) : Personne(nom, prenom, "enseignant(e)-
chercheur"), _type_formation(type_formation), _laboratoire(laboratoire),
_bureau(bureau) {}

        Enseignant(string nom,
                  string prenom,
                  string type_formation,
                  string bureau) : Personne(nom, prenom, "professeur(e)
agrégé(e)"), _type_formation(type_formation), _laboratoire(),
_bureau(bureau) {}

        virtual void affiche(void);

        virtual void profession();
        virtual void qualite();

};

#endif
```

Implementation file

```
#include "header/Enseignant.h"

void Enseignant::affiche(){
    Personne::affiche();
    cout << "et je suis enseignant" << endl;
}

void Enseignant::profession(){
    Personne::qualite();
    if (_laboratoire.empty())
        cout << " en " << _type_formation << " et mon bureau est le " << 
_bureau << endl;
    else
        cout << " en " << _type_formation << " au laboratoire " << 
_laboratoire << " et mon bureau est le " << _bureau << endl;
}

void Enseignant::qualite() {
    Personne::qualite();
    cout << endl;
}

void Enseignant::afficherPersonne(ostream& os) {
    Liste<string>::ListeIterateur it(_experiences);

    Personne::afficherPersonne(os);
    os << "Type de formation : " << _type_formation << endl;
    os << "Bureau : " << _bureau << endl;
    if (!_laboratoire.empty())
        os << "Laboratoire : " << _laboratoire << endl;

    os << "Expériences : " << endl;
    while(it.hasNext())
        os << "- " << it.get() << endl;
}
```

Administratif

Header file

```
#ifndef __ADMINISTRATIF_H
#define __ADMINISTRATIF_H

#include "Personne.h"

using namespace std;

class Administratif : public Personne {
    private :
        // donnees specifiques a administratif
        string _fonction;
                string _bureau;
                virtual void afficherPersonne(ostream&);
    public :
        Administratif(string prenom,
                string nom,
                string fonction,
                string bureau) : Personne(nom, prenom, "personnel
administratif"), _fonction(fonction), _bureau(bureau) {}

        virtual void affiche(void);

        virtual void profession();
        virtual void qualite();
};

#endif
```

Implementation file

```
#include "header/Administratif.h"

void Administratif::affiche(){
    Personne::affiche();
    cout << "et je suis un personnel administratif" << endl;
}

void Administratif::profession(){
    Personne::qualite();
    cout << " en tant que " << _fonction <<  " et mon bureau est le " <<
_bureau << endl;
}

void Administratif::qualite() {
    Personne::qualite();
    cout << endl;
}

void Administratif::afficherPersonne(ostream& os) {
    Liste<string>::ListeIterateur it(_experiences);
    Personne::afficherPersonne(os);
    os << "Fonction : " << _fonction << endl;

    os << "Expériences : " << endl;
    while(it.hasNext())
        os << "- " << it.get() << endl;
}
```

BDD

Header file

```
#ifndef BDD_H
#define BDD_H

#include <iostream>
#include <string>
#include "ListeRecursif.h"
#include "Personne.h"

using namespace std;


class BDD : protected Liste<Personne*> {
public:
    void ajouter(Personne* p);

    void afficheContenuBDD();
};

#endif // BDD_H
```

Implementation file

```
#include "header/bdd.h"

void BDD::ajouter(Personne* p) {
    Liste::ajouter(p, longueur());
}

void BDD::afficheContenuBDD() {
    // Parcourir la liste des personnes et afficher leurs informations
    Liste<Personne*>::ListeIterateur it(*this);
    Personne* cur_pers = NULL;
    int i = 0;
    cout <<
"***************************************************************************" << endl;
    cout << "**************** Database UTBM : ************************************************" << endl;
    cout << "***************************************************************************" << endl << endl;
    while(it.hasNext()){
        cout << "\tEntrée " << ++i << " :" << endl << endl;
        cur_pers = it.get();
        cout << *cur_pers << endl;
    }
    cout << "***************************************************************************" << endl;
    cout << "**************** End of DATABASE ************************************************" << endl;
    cout << "***************************************************************************" << endl << endl;

}
```

Client file

```
#include <iostream>
#include "header/Etudiant.h"
#include "header/Enseignant.h"
#include "header/Administratif.h"
#include "header/bdd.h"

using namespace std;

int main()
{
    /* Populate the database */
    BDD bdd;

    Personne* etd1 = new Etudiant("Jean", "Pierre", "Info", "CS12");

    bdd.ajouter(etd1);

    Personne* etd2 = new Etudiant("Alexandre","Viala","Info");

    bdd.ajouter(etd2);
    etd2->ajouterExperience("ST10 chez Exxelia");
    etd2->ajouterExperience("ST40");

    Personne* etd3 = new Etudiant("Amelie ", "Rodriguez", "Info", "CS13");

    bdd.ajouter(etd3);
    Personne* ensg1 = new Enseignant("Lewis", "Hamilton", "FISE" , "CIAD
Lab" , "M405");

    bdd.ajouter(ensg1);

    Personne* ensg2 = new Enseignant("Alexis", "Flech", "TC", "P101");

    ensg2->ajouterExperience("Classe préparatoire");
    ensg2->ajouterExperience("Enseignant de mathématiques");
    bdd.ajouter(ensg2);

    Personne* admin1 = new Administratif("Elanor", "Courtney", "IT-
Support", "H023");

    bdd.ajouter(admin1);

    /* Display the database */

    bdd.afficheContenuBDD();

    /* Test of the "affiche()" method */
    etd1->affiche();
    etd2->affiche();
    etd3->affiche();
    ensg1->affiche();
    ensg2->affiche();
    admin1->affiche();

    cout << endl;

    /* Test of the "profession()" method */
    etd1->profession();
    etd2->profession();
    etd3->profession();
    ensg1->profession();
    ensg2->profession();
    admin1->profession();

    cout << endl;

    /* Test of the "qualite()" method */
    etd1->qualite();
    etd2->qualite();
    etd3->qualite();
    ensg1->qualite();
    ensg2->qualite();
    admin1->qualite();

    cout << endl;

    /* Test of the "experience()" method */
    etd1->experience();
    etd2->experience();
    etd3->experience();
    ensg1->experience();
    ensg2->experience();
    admin1->experience();

    cout << endl;

    /* Test of the "getNom()" method */
    cout << etd1->getNom() << endl;
    cout << etd2->getNom() << endl;
    cout << etd3->getNom() << endl;
    cout << ensg1->getNom() << endl;
    cout << ensg2->getNom() << endl;
    cout << admin1->getNom() << endl;



    return 0;
}
```

Compilation

To compile, I used the qmake software. It is a software developped by the Qt Company which let us generate a makefile from a `.pro` file, easier to write than a full makefile. **0_bdd_personnes.pro :**

```
TEMPLATE = app
CONFIG += console c++11
CONFIG -= app_bundle
CONFIG -= qt

INCLUDEPATH += header

TARGET = ../tp3

SOURCES += \
        main.cpp \
        Personne.cpp \
        Administratif.cpp \
        Etudiant.cpp \
        Enseignant.cpp \
        bdd.cpp

HEADERS += \
        ListeRecursif.h \
        Personne.h \
        Administratif.h \
        Etudiant.h \
        Enseignant.h \
        bdd.h
```

To compile, simply run :

```
qmake ; make
```

or simply `make` if the makefile has already been generated.

To erase all the unnecessary file run :

```
make distclean
```

or only `make clean` if you want to keep the makefile and the qmake stash.