PoP Série 2 Solution

H1 :
1) Lecture/écriture de fichier
2) type concret et test unitaire d’un module

Exercice 1.3 (niveau 2): extraction de données d’un fichier texte

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>

using namespace std;

enum {XY, YZ, ZX};

int main(int argc, char *argv[])
{
   if (argc < 4)
   {
      // les 3 chaînes sont concaténées automatiquement
       cout << "Usage incorrect\n Usage: parser_obj"
               " [source_file] [dest_file] [code extraction]\n"
               "Code extraction:\n0: XY\n1: YZ\n2: ZX\n";
      exit(0);
   }

   ifstream fentree ;
   fentree.open(argv[1]);
   if (fentree.fail())
   {
      cout << "Impossible d'ouvrir le fichier: "
           <<  argv[1] << endl;
      exit(0);
   }

   ofstream fsortie;
   fsortie.open(argv[2]);
   if (fsortie.fail())
   {
      cout << "Impossible d'ouvrir le fichier: "
           << argv[2] << endl;
      exit(0);
   }
```

```
   int code_extraction(stoi(argv[3])); //sem1 Topic 13 slide15

   // Extrait les lignes commençant par v du fichier Obj
   //
   string line;
   char v;
   double x, y, z;

   while(getline(fentree,line))
   {
      istringstream data(line) ;
      if(!(data >> v >> x >> y >> z))
        continue;

      if (v =='v') // Ecrit la ligne dans le fichier fsortie
      {
        switch (code_extraction)
        {
        case XY:
           fsortie << x << " " << y << endl ;
           break;

        case YZ:
           fsortie << y << " " << z << endl ;
           break;

        case ZX:
           fsortie << z << " " << x << endl ;
           break;
       }
     }
   }
   fentree.close();
   fsortie.close();

   return EXIT_SUCCESS;
}
```

Exercice 2.2 (niveau 1): module exportant un type concret

0) Que met-on dans *l'interface* **etudiant.h** ?

<u>Première conséquence</u> : la définition détaillée du type Etudiant doit être dans **etudiant.h**. Pourquoi ?

Pour que le compilateur puisse *calculer l'espace mémoire* nécessaire pour mémoriser l'état d'une variable. Ce calcul est fait à chaque déclaration de variable.

<u>Deuxième conséquence</u> : **etudiant.h** doit être inclut dans **etudiant.cc.** Pourquoi ?

En fait, ce qui est montré dans l'interface a toutes les raisons d'être aussi utile dans l'implémentation, comme par exemple la définition détaillée du type Etudiant. On pourrait dupliquer le contenu de **interface.h** pour l'avoir aussi dans **etudiant.cc**. Cependant, à chaque modification de **interface.h**, il faut aussi mettre à jour « manuellement » **interface.cc**. Sans parler des risques d'incohérence entre les deux fichiers, c'est une pure perte de temps comparé à la solution « automatique » d'inclure l'interface **etudiant.h** dans l'implémentation **etudiant.cc**.

1) *interface* **etudiant.h** :

```
#ifndef ETUDIANT_H
#define ETUDIANT_H
#include <string>

#define MAX_SECTION 2

struct Etudiant
{
	string nom;
	string prenom;
	int  age;
	string section;
};

Etudiant etudiant_init(void);
bool   etudiant_are_the_same(Etudiant e1, Etudiant e2);
void   etudiant_affiche(Etudiant e);

#endif
```

2) *implémentation* **etudiant.cc** : définir les fonctions déclarées dans *l'interface*.

```
// fichier : etudiant.cc
// version : 1.0
//
#include <iostream>
#include <limits>

using namespace std;
#include "etudiant.h"

Etudiant etudiant_init(void)
{
	Etudiant e;

	cout << "\nNom de famille : ";
	cin  >> e.nom ;

	cout << "\nPrénom : ";
	cin  >> e.prenom ;

	bool echec(false);
	do
	{
		echec = false;
```

```
            cout << "\nAge: ";
            if(not(cin >> e.age))
            {
                  echec = true;
                  cin.clear();
                  cin.ignore(numeric_limits<streamsize>::max(),'\n');
            }
      }while(echec || e.age <= 0);

    do
    {
          cout << "\nsection (max "<< MAX_SECTION << " caractères): ";
          cin  >> e.section;
    }while(e.section.size() != MAX_SECTION) ;

    return e;
}

bool etudiant_are_the_same(Etudiant e1, Etudiant e2)
{
    return(e1.nom    == e2.nom    &&
           e1.prenom == e2.prenom &&
           e1.age    == e2.age    &&
           e1.section== e2.section);
}

void   etudiant_affiche(Etudiant e)
{
      cout << e.nom << " " << e.prenom << " " << e.age
           << " "  << e.section << endl;
}
```

Le module etudiant est, autant que possible, responsable de la validité des données mémorisées dans le type Etudiant. Cela est surtout visible dans la fonction qui initialise les valeurs des champs. Ce code ajoute des tests et éventuellement refuse des données invalides :
- on prend seulement MAX_SECTION pour la section
- on veut un age strictement positif

3) test unitaire du module **etudiant**: écrire un module **test** (limité à une implémentation)

```
// fichier : test.cc
// version : 1.0
//
#include <cstdlib>
#include <iostream>
#include <vector>
#include <limits>

using namespace std;
#include "etudiant.h"

bool test_doublon(int n, vector<Etudiant>& tab, Etudiant e)
{
      int i;
      for(i=0 ; i<n ; i++)
      {
```

```
            if(etudiant_are_the_same(tab[i],e))
                  return true;
      }
      return false;
}

int main(void)
{
      int nb_etudiant=0;

      bool echec(false);
      do
      {
          echec = false;
          cout << "\n nombre d'étudiants: ";
          if(not(cin >> nb_etudiant))
          {
              echec = true;
              cin.clear();
              cin.ignore(numeric_limits<streamsize>::max(),'\n');
          }
      }while(echec || nb_etudiant <= 0);

      vector<Etudiant> tab(nb_etudiant);

      for(int i=0 ; i < nb_etudiant ; i++)
      {
          Etudiant temp ;
          bool doublon = false;
          do
          {
              doublon = false;
              cout << "\nlecture etudiant d'indice " << i << " :";
              temp = etudiant_init();
              if((doublon = test_doublon(i, tab, temp)))
                  cout << "erreur: étudiant déjà présent" << endl;
          }
          while(doublon);

          tab[i] = temp;
      }

      cout << "\nAffichage du contenu du tableau d'étudiants" << endl;

      for(Etudiant etudiant : tab)
            etudiant_affiche(etudiant);

      cout << "Fin du programme" << endl;

      return EXIT_SUCCESS;

}
```

Pour ce projet simple, on peut compiler avec : **`g++ test.cc etudiant.cc –o test`**

4) Que se passe-t-il à la compilation si on fournit seulement une prédéclaration du modèle de structure au lieu de la définition complète de ce modèle de structure dans **etudiant.h** (on cache la définition du modèle de structure `Etudiant` dans **etudiant.cc)** ?

Le compilateur va signaler une erreur dès la première déclaration de variable ou de tableau dans le module **test.cc** car ce module ne dispose plus de la description détaillée du type Etudiant. La seule chose à disposition est un nom de type, pas plus. On peut simplement déclarer des pointeurs avec ce nom de type, rien de plus. Le détail du modèle de structure doit être dans le fichier d'interface pour les types concrets.

<u>5) Remarque finale </u>:

La faiblesse du type concret Etudiant est que rien n'empêche le code de **test.cc** d'accéder aux champs de Etudiant et d'y mettre des valeurs incorrectes ; le module **etudiant** perd le « contrôle qualité » sur le type qu'il met à disposition.

En résumé les types concrets sont ok pour des types utilitaires de bas-niveau stables et bien maitrisés. Le projet impose d'utiliser des classes pour un certain nombre de types afin de bénéficier de ses propriétés d'encapsulation qui permettent aux modules de préserver leur contrôle sur leurs données.