

PoP Série 3 niveau 0

Usage de static à l'échelle d'un module

La série précédente niveau 0 a déjà illustré l'usage de **static** à l'échelle de variables locales (automate de lecture de fichier). Nous examinons ici l'usage de **static** à l'échelle d'un petit module pour gérer une information partagée à l'échelle de ce module (et invisible à l'extérieur du module). Cet usage correspond à l'exploitation de l'espace de noms non-nommé de ce module.

Exercice 1.1 : module student gérant un ensemble de variables Student

La structure Student n'est autre que la structure Etudiant déjà rencontrée dans la série2 exercice 2.2 sur le test unitaire d'un **type concret**. Voici l'interface du module student.h :

```
#ifndef STUDENT_H
#define STUDENT_H
// version : 1.1
#include <string>

#define MAX_SECTION 2

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

// keep asking correct data until a correct student is initialized
Student student_init(void);

// return true is successfully added
// and false if already present in database
bool student_add_to_database(const Student& new_student);

// return true if all fields of e1 and e2 are equal
bool student_are_the_same(const Student& e1, const Student& e2);

// display all students form the database hidden in implementation
void student_display_all();

#endif
```

La nouveauté est que le module student tient à jour un **vector<Student>** d'un ensemble de variables Student et que celui-ci n'est visible **que** dans l'implémentation **student.cc**

C'est une différence notable avec une variable de classe qui elle est visible dans l'interface de la classe (et serait dans student.h). L'approche avec une variable static à un module diminue

les dépendances induites par l'interface. En effet la personne responsable du module peut changer son choix pour mémoriser l'ensemble de Students sans induire de changement pour les modules de niveau supérieurs. Par exemple on pourrait changer vector en une liste chaînée (sera vu plus tard) sans devoir recompilier les modules de niveau supérieur. Ça n'est pas le cas pour une variable de classe car le changement sera visible dans l'interface du module et donc impliquera une recompilation des modules de niveau supérieurs, même s'ils n'ont pas accès à la variable de classe.

A l'initialisation ce vector est vide comme le montre sa déclaration dans le module student.cc :

```
// fichier : student.cc
// version : 1.1
#include <iostream>
#include <limits>
#include <vector>
```

```
using namespace std;
#include "student.h"
```

```
static    vector<Student> tab; // vide à l'initialisation
```

Ensuite à chaque appel de la nouvelle fonction student_add_to_database() on y ajoute un nouvel élément sauf s'il est déjà présent dans ce tableau dynamique.

```
bool    student_add_to_database(const Student& new_student)
{
    for(const Student& e: tab)
    {
        if(student_are_the_same(e, new_student))
            return false;
    }

    tab.push_back(new_student);
    return true;
}
```

Cette fonction utilise la fonction de test d'égalité de deux variables Student puisqu'on ne peut pas utiliser l'opérateur == sur des variables struct (il faudrait surcharger cet opérateur ; cf série3 MOOC) :

```
bool student_are_the_same(const Student& e1, const Student& e2)
{
    return(e1.nom    == e2.nom    &&
           e1.prenom == e2.prenom &&
           e1.age    == e2.age    &&
           e1.section== e2.section);
}
```

A tout moment la fonction `student_display_all()` peut afficher l'ensemble des Students présent dans le vector. Celui-ci étant unique et accessible par la fonction dans le module `student`, il n'est pas nécessaire de préciser de paramètre :

```
// the special character '\t' is a tabulation
void student_display_all()
{
    for(const Student& e: tab)
    {
        cout << e.nom << " \t" << e.prenom << " \t" << e.age
             << " \t" << e.section << endl;
    }
}
```

D'autres fonctions d'exploitation du `vector<Student>` pourraient être proposées par ce module pour réaliser toutes sortes de tâches ; l'avantage de cette approche avec le tableau dynamique en `static` est que le module dispose de toute l'information de manière efficace et sécurisée car le tableau dynamique est invisible à l'extérieur du module. On limitera ce type d'usage de `static` à un module gérant un seul type de donnée (struct ou class) pour éviter la perte d'encapsulation qui en résulterait si plusieurs types sont gérés dans le même module.

Enfin, le module **`student.cc`** est utilisé à un niveau supérieur par un module **`student_test.cc`** qui peut représenter une application mais ici c'est seulement pour effectuer des tests en vue de valider les fonctions offertes par l'interface **`student.h`**.

```
// fichier : student_test.cc
// version : 1.1
//
#include <cstdlib>
#include <iostream>
#include <vector>
#include <limits>

using namespace std;
#include "student.h"

int main(void)
{
    int nb_student=0;

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

```

    }
    }while(echec || nb_student <= 0);

int i(0);
while(i< nb_student)
{
    Student temp(student_init());

    if(student_add_to_database(temp) ++i;
    else
        cout << "This student is already in the database !"
            << endl;
}

cout << "\nStudent database content\n" << endl;

student_display_all();

cout << "\nEnd of program" << endl;

return EXIT_SUCCESS;
}

```

Le code est disponible dans le fichier archive PoP_s3_niv0_code.