PoP Série 2 niveau 0

Première partie : lecture/écriture de fichier

Exercice 1.1: Automate de lecture d'un fichier de configuration

On veut écrire un programme qui lit un fichier répondant au **format** indiqué pour un fichier de configuration structuré comme suit :

- les lignes vides ou commençant par # (commentaire) sont ignorées

- le fichier comporte 3 sections sur le même modèle :
    - tout d'abord un nombre entier positif indique un certain nombre d'entités N
    - puis une liste des entités est fournie, avec une entité par ligne, si N >0
    - chaque entité peut avoir un format de lecture différent

Pour les livreurs : un nom suivi d'un entier interprété comme un booléen
Pour les véhicules : un entier d'identification suivi d'un entier interprété comme un booléen
Pour les livraisons : un nom de livreur suivi d'un entier identifiant un véhicule

L'automate de lecture prévoit tous les cas possibles de lecture selon les nombres d'entités : on appelle ces cas des « **états** » de lecture. Le cours montre les conditions à remplir pour passer d'un état à un autre ; la représentation sous forme d'un graphe est très utile pour mettre au point le programme.

L'architecture de ce programme repose sur deux modules :

- Module de haut niveau : responsable de la vérification de la structure du fichier et des nombres d'entités. Il met en œuvre le principe d'abstraction pour décomposer la lecture en plusieurs niveaux de vérifications. Il délègue au module **datastruct** les vérifications spécifiques à chaque entité isolée.
- Module **datastruct** : responsable des classes Livreur, Vehicule et Livraison.

Commençons par examiner les responsabilités de **datastruct** :

- Principe d'encapsulation : les attributs sont `private`
- Pour cet exemple, on n'a seulement besoin d'un *constructeur* pouvant aussi servir de constructeur par défaut, et surtout d'une méthode de *lecture* des données d'une entité à partir d'un paramètre de type `istringstream`. Ces deux méthodes sont responsables de vérifier le domaine de validité des paramètres ou des valeurs lues ; ici on prévoit seulement l'espace pour cela sans entrer dans les détails. La seconde vérification faite par la méthode de lecture est de s'assurer qu'on a bien pu lire les attributs dans le `istringstream`. Noter le type booléen de la méthode de lecture qui permet de signaler au niveau appelant le succès ou l'échec de l'opération.

Le module de niveau supérieur **automate_lecture_fichier** inclut l'interface **datastruct.h** pour pouvoir créer les structures de données dont il est responsable. Ici il s'agit d'un `vector` pour chaque type d'entité (lignes 24-26). Ces déclarations sont préfixées par `static` pour rendre

ces éléments confidentiels au module ; elles appartiennent à l'espace de nom non-nommé du module. On remarque deux autres déclarations `static` confidentielles au module lignes 20-21 ; elle servent pour la gestion de l'automate de lecture car celle-ci est répartie entre plusieurs fonctions.

Les lignes 12-14 et 17 déclarent des constantes entières avec `enum` pour mieux documenter les différents états de l'automate de lecture (ligne 17) et les erreurs détectées à ce niveau (lignes 12-14).

Le principe d'abstraction répartit les vérifications de la manière suivante :

- **main** récupère un nom de fichier sur la ligne de commande. Il procède à deux appels de lecture du fichier. Le second appel est destiné à s'assurer que l'automate de lecture est bien ré-initialisé avant chaque nouvelle lecture ; **main**() dispose du booléen renvoyé par la fonction de lecture pour savoir si celle-ci s'est bien passée et fait afficher un message de feedback.

- **lecture :** appelle tout d'abord **reset**() qui vide les structures de données (les vectors) et ré-initialise les variables `static` qui gèrent l'automate de lecture. Ensuite elle vérifie le succès d l'ouverture du fichier et le lit ligne par ligne jusqu'à la fin de fichier en éliminant les commentaires. Chaque ligne qui n'est pas un commentaire est convertie en `istringstream` pour le décodage délégué à la fonction **decodage_ligne** qui gère l'automate de lecture en faisant la détection des erreurs simples (détectable au niveau d'une entité isolée). Par contre, c'est bien dans la fonction **lecture** qu'il est possible d'effectuer des vérifications complexes entre les ensembles d'entités juste après la boucle `while`. Le booléen renvoyé par lecture indique le succès/échec de la lecture.

- **decodage_ligne** est le véritable automate de lecture. Il décode la ligne courante qui lui est envoyée par **lecture**(). Cette fonction fait apparaitre l'ensemble des états de l'automate et délègue à une fonction le traitement de la ligne lue selon l'état courant. Elle est responsable de vérifier que l'état courant est correct. Elle renvoie un booléen indiquant le succès/l'échec du traitement de la ligne reçue.

- on trouve 6 fonctions après decodage_ligne, chacune étant responsable du décodage de la ligne lue dans le fichier pour l'un des états de l'automate de lecture. Pour ce programme, le module de haut niveau est responsable de la construction des vectors d'entité, ce qui en fait le niveau responsable de la vérification de la lecture correcte des nombres d'entité. Par contre, on délègue aux méthodes de lecture du module **datastruct** lorsqu'il faut décoder une ligne de fichier contenant les données d'une entité Livreur ou Vehicule ou Livraison.

La dernière fonction **erreur**() rassemble tous les cas d'erreurs considérés au niveau du module de haut niveau. On retrouve les symboles définis avec `enum` en début de fichier. On peut commenter l'instruction exit qui fait quitter le programme si on préfère poursuivre l'exécution avec l'information du booléen de succès/échec de la lecture.

Le code complet se trouve dans le fichier archive du code source sous le nom **automate** avec 3 fichiers de configuration **configX.txt** que vous pouvez modifier pour tester le bon fonctionnement du programme.

Exercice 1.2 (niveau 0): Lire et afficher le contenu d'un fichier texte

On veut écrire un programme **afficher.cc** qui permet de lire et d'afficher le contenu d'un fichier texte avec le nom **texte.txt** qui se trouve dans le même répertoire que **afficher.cc**

Cet exercice reprend pas à pas les différentes étapes pour y parvenir

1. Ouvrez le fichier (vide) **afficher.cc** dans votre éditeur pour y parvenir.
2. Préparez le "coquille vide" de base accueillant votre programme. Pour utiliser des fichier, il faut inclure **iostream** et **fstream**

```
#include <cstdlib>

int main()
{
    return EXIT_SUCCESS;
}
```

3. On va utiliser la fonction **get()** pour lire le fichier et renvoyer l'octet du caractère lu. Cette fonction renvoie la valeur **EOF** quand on arrive à la fin du fichier. On va se servir de cette particularité pour arrêter la lecture à la fin du fichier. Cette fonction a besoin d'une variable **ifstream**.

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

int main()
{
    ifstream fichier("texte.txt");
    if(fichier.fail())
    {
        cout << "Le fichier texte est absent" << endl;
        return EXIT_FAILURE;
    }
    char octet;
    while((octet = fichier.get()) != EOF)
    {
       cout.put(octet);
    }

    fichier.close();

    cout << "\nFin du fichier texte" << endl;

    return EXIT_SUCCESS;
}
```

Seconde partie : prévenir l'inclusion multiple

Nous allons illustrer un exemple d'inclusions multiples (cours sur le préprocesseur). De plus les fonctions indiquées dans l'interface des modules seront implémentées sous formes de ***stub***, ce qui permettra de découvrir cette forme minimale de définition d'une fonction.

Exercice 2.1 (niveau 0): pathologie de l'inclusion multiple

Pour simplifier, nous allons travailler avec des modules quasiment vides, l'important étant de se concentrer sur les conséquences de l'inclusion multiple et la manière de résoudre ce problème. L'architecture logicielle de notre exemple est illustrée ci-dessous. Elle implique les quatre modules suivants, du bas niveau au plus haut niveau : nom, fichier1, fichier2, fichier3.

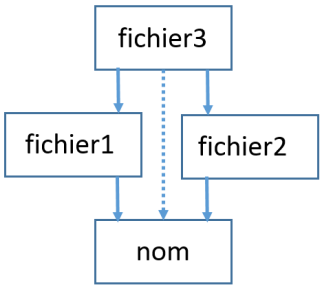

Fig 1 : architecture logicielle

Tout d'abord, pour cet exercice, les noms des fonctions exportées dans les interfaces des modules respectent une convention pour éviter la collision des noms de fonctions[1] :

- Le nom d'une fonction exportée d'un module est préfixé par le nom du module. Nous aurions pu aussi choisir de définir un espace de nom distinct pour chaque module.

2.1.1) Le problème de l'inclusion multiple

Supposons que le module **nom** exporte et un modèle de structure et une seule fonction dans **nom.h**. Pour qu'elle soit compilable, on doit aussi définir le symbole MAX_NOM, ce qui donne comme version minimum (non-robuste) du contenu du fichier **nom.h** :

```
#define MAX_NOM 30
struct Nom
{
      char tab[MAX_NOM] ;
};
void nom_lecture(Nom& val);
```

Du coté de l'implémentation **nom.cc**, il faut au minimum inclure le fichier **nom.h** car les informations qu'il contient sont aussi utiles pour définir les fonctions. Dans l'exemple de cet exercice on ne s'intéresse pas aux détails des instructions de la fonction nom_lecture. CEPENDANT nous voulons qu'elle soit définie dans une version minimale qui compile sans

[1] le problème de collision des noms ne concerne pas les méthodes d'une classe qui disposent de leur propre espace de nom

produire d'erreur. On appelle cela un *stub*. Dans le cas d'une fonction qui ne renvoie rien, il suffit d'indiquer un bloc vide qui correspond à l'instruction nulle (qui ne fait rien) :

```
#include "nom.h"

void nom_lecture(Nom& val)
{
}
```

Le module **nom** peut déjà être compilé séparément et produire un fichier **nom.o** avec :

```
g++ -std=c++11 -Wall nom.cc –c
```

Voyons maintenant le reste de l'architecture. Supposons que les modules fichier1 et fichier2 exportent chacun une fonction qui utilise aussi le symbole MAX_NOM. Cette dépendance impose d'inclure **nom.h** à la fois dans fichier1.h et fichier2.h pour disposer de ce symbole.

Voici fichier1.h

```
#include "nom.h"

void fichier1_test1(char nom[MAX_NOM]);
```

Et son implémentation fichier1.cc avec un stub :

```
#include "fichier1.h"

void fichier1_test1(char nom[MAX_NOM])
{
}
```

Et voici fichier2.h

```
#include "nom.h"

void fichier2_test2(char nom[MAX_NOM]);
```

Et son implémentation fichier2.cc avec un stub :

```
#include "fichier2.h"

void fichier2_test2(char nom[MAX_NOM])
{
}
```

A ce stade de nos définitions, on peut aussi compiler séparément **fichier1.cc** et **fichier2.cc** et obtenir sans problème **fichier1.o** et **fichier2.o** .
Remarque : S'il n'y avait pas eu cette dépendance de **fichier1.h** et **fichier2.h** vis-à-vis de **nom.h** on aurait plutôt inclut **nom.h** dans **fichier1.cc** et **fichier2.cc** car cela crée moins de dépendances. En effet, le problème de l'inclusion multiple va apparaitre au niveau supérieur de **fichier3.cc** qui va inclure à la fois **fichier1.h** ET **fichier2.h.** Ce faisant il y a deux fois l'inclusion de **nom.h** et deux conséquences :

- Une dépendance indirecte de **fichier3** vis-à-vis de **nom** (flèche avec trait en pointillé dans le dessin de l'architecture logicielle)
- Des erreurs de compilation pour **fichier3.cc** à cause des doubles définitions du modèle de structure et de la fonction de nom.h

Supposons que le fichier3.cc contienne au moins le stub de la fonction main():

```
#include <cstdlib>
#include "fichier1.h"
#include "fichier2.h"

int main(void)
{
    return EXIT_SUCCESS ;
}
```

La compilation séparée de fichier3.cc produit sa première erreur quand il y a inclusion de fichier2.h car cette inclusion va elle-même produire la seconde inclusion de nom.h et ainsi produire les erreurs de double-définition.

2.1.2) La solution de l'inclusion multiple

Tout fichier en-tête (.h) destiné à être inclus dans un module d'une application doit être encadré par trois directives :

```
#ifndef  NOM_DE_SYMBOLE_UNIQUE
#define  NOM_DE_SYMBOLE_UNIQUE

Ici le contenu utile du fichier .h

#endif
```

Où **NOM_DE_SYMBOLE_UNIQUE** est un symbole qui ne doit pas exister ailleurs dans l'application. Une approche systématique est de choisir un nom de symbole qui est simplement le nom du fichier .h en majuscules. Dans notre exemple, cela donne :

```
#ifndef  NOM_H
#define  NOM_H

#define MAX_NOM 30
struct Nom
{
    char tab[MAX_NOM]
};
void nom_lecture(struct Nom val);

#endif
```

Pourquoi cela permet-il d'éviter le problème des doubles définitions ?

Revenons à **fichier3.cc** et mettons-nous à la place du compilateur quand il inclut les fichiers d'interface un à un. Nous allons littéralement copier-coller le contenu du fichier d'interface pour suivre à la lettre l'action d'inclure un autre fichier :

Situation après l'include de **fichier1.h** qui lui-même a inclus le contenu de **nom.h** car le symbole NOM_H n'étant pas défini à ce moment là, la directive **ifnedf** était VRAI et a autorisé d'inclure tout ce qui se trouve entre **ifndef** et **endif** :

```
… contenu de stdlib.h …
#define  NOM_H

#define MAX_NOM 30
struct Nom
{
    char tab[MAX_NOM] ;
};
void nom_lecture(struct Nom val);

void fichier1_test1(char nom[MAX_NOM]);

#include "fichier2.h"

int main(void)
{
    return EXIT_SUCCESS ;
}
```

Ensuite vient l'include de **fichier2.h** qui lui-même commence par une directive d'include de **nom.h.** CEPENDANT cette fois-ci le symbole NOM_H a été défini précédemment, donc la directive **ifnedf** donne FAUX et interdit d'inclure tout ce qui se trouve entre **ifndef** et **endif.** Il n'y a donc pas de double définition:

```
… contenu de stdlib.h …
#define  NOM_H

#define MAX_NOM 30
struct Nom
{
    char tab[MAX_NOM];
};
void nom_lecture(struct Nom val);
void fichier1_test1(char nom[MAX_NOM]);
void fichier2_test2(char nom[MAX_NOM]);

int main(void)
{
    return EXIT_SUCCESS ;
}
```

Exercice : créer les fichiers sources décrits plus haut, faire la compilation séparée dans le cas pathologique pour voir le type d'erreur indiqué par g++. Ensuite, adopter l'organisation proposée au moins pour **nom.h**. Compiler séparément puis produire l'exécutable et lancer le pour voir que les stub ne produisent pas de problème à l'édition de liens et à l'exécution.

Conclusion : il faut systématiquement adopter l'organisation avec **ifndef / define / endif** pour tous les fichiers en-têtes, peu importe le nombre de fois où ils sont inclus dans d'autres modules. Prendre garde d'utiliser un symbole DIFFERENT pour la directive **ifndef** pour chaque fichier .h.