PoP C++ Série 3 niveau 0

Première partie : lecture/écriture de fichier

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

On veut écrire un petit 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, on lit un nom suivi d'un entier pour un booléen
Pour les véhicules, on lit un entier d'identification suivi d'un entier pour un booléen
Pour les livraisons, on lit 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.

Le programme est organisé à l'aide de 4 fonctions :
- **main** récupère un nom de fichier sur la ligne de commande
- **lecture** ouvre le fichier et le lit ligne par ligne et éliminant les lignes inutiles
- **decodage_ligne** est le véritable automate de lecture. Il décode la ligne courante qui lui est envoyée par lecture().

A propos de static : On remarque quelques variables locales **static** dans le code présenté ici. Le fait d'ajouter ce mot clef à la déclaration d'une variable locale la rend permanente, c'est-à-dire que la variable mémorise sa valeur d'un appel au suivant. Sa portée n'est pas modifiée ; elle reste locale.

- une fonction **erreur** centralise le traitement des erreurs avec l'affichage d'un message puis l'arrêt du programme. Pour cette fonction on a créé une liste de symboles à l'aide du mot clef **enum** :

```
// symboles et type pour codes d'erreurs
enum Erreur_lecture {LECTURE_ARG, LECTURE_OUVERTURE,
                     LECTURE_NB0, LECTURE_LIVREUR,
                     LECTURE_NB1, LECTURE_VEHICULE,
                     LECTURE_NB2, LECTURE_LIVRAISON,
                     LECTURE_FIN, LECTURE_ETAT};
```

Voici les prototypes des 3 fonctions en plus de main. Le type enuméré est utilisé pour le paramètre de la fonction **erreur**.

```
void lecture(char * nom_fichier);
void decodage_ligne(string line);
void erreur(Erreur_lecture code);
```

La fonction **main** délègue le travail principal à la fonction **lecture**

```
// lit le fichier dont le nom est transmis sur la ligne de commande
int main(int argc, char * argv[])
{
  if(argc != 2) erreur(LECTURE_ARG);

  lecture(argv[1]);

  return EXIT_SUCCESS;
}
```

La fonction **lecture** se charge ***d'ouvrir*** le fichier et de ***filtrer les lignes inutiles*** (vides, commentaires). une ligne utile est transmise à la fonction **décodage_ligne** qui réalise l'automate de lecture.

```
// traite le fichier ligne par ligne.

void lecture( char * nom_fichier)
{
    string line;
    ifstream fichier(nom_fichier);
    if(!fichier.fail())
    {
        // l'appel de getline filtre aussi les séparateurs
        while(getline(fichier >> ws,line))
        {
           // ligne de commentaire à ignorer, on passe à la suivante
           if(line[0]=='#')  continue;

          decodage_ligne(line);
        }
        cout << "fin de la lecture" << endl;
    }
    else erreur(LECTURE_OUVERTURE);
}
```

La fonction **décodage_ligne** réalise l'automate de lecture qui vérifie la validité du format. Elle commence par déclarer un **input string stream** nommé **data** qui est initialisé avec la string **line** envoyée en paramètre (faire **#include <sstream>**). C'est ce string stream qui est décodé selon l'état courant mémorisé dans la variable **static etat**.

La variable **etat** est initialisée une seule fois au lancement du programme avec l'état initial **NB0** ; on remarquera que **NB0** provient d'une autre liste de symboles aussi créée avec le mot clef **enum**. Cette manière de créer des symboles est beaucoup plus compacte qu'une longue liste de directives define.

la variable compteur **i** et le **total** sont aussi en **static** pour les mémoriser d'un appel au suivant.

```
// décodage selon l'etat courant d'une ligne lue dans le fichier
// met à jour l'etat

void decodage_ligne(string line)
{
  istringstream data(line);

  // états de l'automate de lecture
  enum Etat_lecture {NB0,LIVREUR,NB1,VEHICULE,NB2,LIVRAISON,FIN};

  static int etat(NB0); // état initial
  static int i(0), total(0);
  int num(0);
  bool disp(false);
  string name;
```

le cœur de l'automate est une instruction **switch** sur la valeur de la variable **etat**. Pour chaque état prévu **data** va être décodé puis analysé de manière distincte. A cette occasion on teste si la lecture se passe bien en vérifiant que **le string stream** renvoie true. La fonction **erreur** est appelée en cas d'erreur.

Une conséquence de l'analyse de la ligne est la possibilité d'un changement de l'état. Par exemple si on se trouve dans l'état initial **NB0** et que la variable **total** lit une valeur **nulle** alors on passe dans l'état **NB1** sinon on passe dans l'état **LIVREUR**. L'appel suivant de cette fonction en tiendra compte grâce à la mémorisation de type **static** de la variable **etat**.

L'autre conséquence de la lecture est une action d'affichage avec cout. Bien sûr dans le cas d'un projet on ajouterait d'autres actions comme la construction d'une structure de données.

```
  switch(etat)
  {
  case NB0:
      if(!(data >> total)) erreur(LECTURE_NB0);
      else i=0 ;
      if(total==0) etat=NB1;
      else         etat=LIVREUR ;
      cout << "Nb de livreurs: " << total << endl;
      break;

  case LIVREUR:
      if(!(data >> name >> disp)) erreur(LECTURE_LIVREUR);
      else ++i;
      if(i == total) etat=NB1 ;
      cout << "Livreur " << i << ": "  << name
                             << " disp : "  << disp << endl;
      break;

  case NB1:
      if(!(data >> total)) erreur(LECTURE_NB1);
      else i=0 ;
      if(total==0) etat=NB2;
      else         etat=VEHICULE ;
      cout << "Nb de véhicules: " << total << endl;
```

```
        break;

    case VEHICULE:
        if( !(data >> num >> disp)) erreur(LECTURE_VEHICULE);
        else ++i;

        if(i == total) etat=NB2 ;
        cout << "Véhicule " << i << ": "  << num
                                << " disp : "  << disp << endl ;
        break;

    case NB2:
        if(!(data >> total)) erreur(LECTURE_NB2);
        else i=0;

        if(total==0) etat=FIN;
        else         etat=LIVRAISON ;
        cout << "Nb de livraisons: " << total << endl;
        break;

    case LIVRAISON:
        if( !(data >> name >> num)) erreur(LECTURE_LIVRAISON);
        else ++i;

        if(i == total) etat=FIN ;
        cout << "Livraison " << i << ": "  << name
                                << " véhicule : "  << num << endl;
        break;

    case FIN: erreur(LECTURE_FIN) ;
        break;

    default: erreur(LECTURE_ETAT);
    }
}
```

La dernière fonction rassemble tous les cas d'erreurs considérés dans ce programme. On retrouve les symboles définis avec enum en début de fichier.
On pourrait préciser les messages pour les symboles LECTURE_NB0 jusqu'à LECTURE_LIVRAISON pour fournir une information plus précise à l'usager.

```
// affiche un message d'erreur puis quitte le programme
void erreur(Erreur_lecture code)
{
    switch(code)
    {
        case LECTURE_ARG : cout << " nom de fichier manquant\n";  break;
        case LECTURE_OUVERTURE: cout << " ouverture impossible\n";break;
        case LECTURE_NB0 :
        case LECTURE_NB1 :
        case LECTURE_NB2 :
        case LECTURE_LIVREUR :
        case LECTURE_VEHICULE :
        case LECTURE_LIVRAISON : cout << " lecture impossible \n";
        case LECTURE_FIN : cout << " format non respecté \n";     break;
        case LECTURE_ETAT : cout << " etat inexistant\n";         break;
        default: cout << " erreur inconnue\n";
    }
    exit(EXIT_FAILURE);
}
```

Le code complet se trouve dans le fichier archive du code source sous le nom **automate_lecture_fichier.cc** avec un fichier de configuration **config1.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.

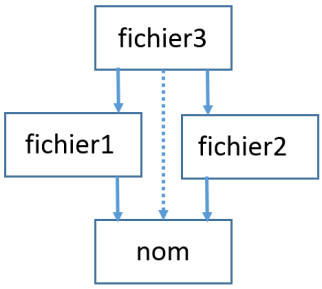

Fig 1 : architecture logicielle

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.

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.

1.1) 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.