PoP C++ Série 3

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

H2 : MOOC [MOOC Introduction à la programmation orientée objet (en C++)]
Série semaine 2: constructeur/ destructeur
disponible sur moodle

Première partie : Lecture/écriture de fichier

Buts

Dans cette partie vous allez voir comment on peut écrire et lire dans un fichier. Ceci est très utile quand on veut par exemple initialiser un programme ou pour sauvegarder des données. L'exemple du cours sur l'automate de lecture est développé dans le document séparé niveau 0.

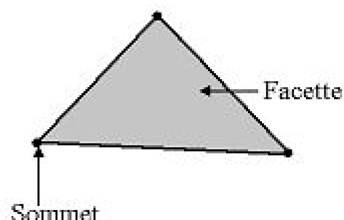
Exercice 1.1 (niveau 0): automate de lecture d'un fichier de configuration

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

---

Exercice 1.3 (niveau 2): Extraction de données d'un fichier texte

En infographie, beaucoup de logiciels utilisent le format "Obj" (extension .obj) pour sauvegarder les sommets et les facettes d'une forme géométrique. Une facette est constituée d'au moins trois sommets comme le montre la figure ci-dessous:



Ce format de fichier contient notamment les lignes suivantes:
Les commentaires commencent par "#"

 # The units used in this file are centimeters.

Les lignes contenant des sommets 3D commencent par "v"
v -0.500000 -0.500000 0.500000

Enfin, les lignes contenant la description de chaque facette commencent par "f"
f 7/13/21 1/1/22 3/3/23 5/14/24

Chaque description de facette contient plusieurs triplets d'indices. Le premier indice d'un triplet représente le numéro du sommet 3D défini précédemment. Les 2 autres indices représentent les coordonnées de texture, les normales....
Cette ligne indique par exemple que la facette est constituée des sommets 7, 1, 3 et 5 (l'ordre des sommets d'une facette est important mais nous n'en discuterons pas ici).

Les autres lignes du fichier indiquent des descriptions des textures, des normales...

1.3.1) Ecrivez un programme qui, étant donné deux noms de fichiers passés en paramètre, analyse le premier et sauvegarde ***les coordonnées des sommets 3D*** dans le second fichier. Testez votre programme avec le fichier "human.obj" présent dans le fichier archive associé à la série.

Conseils :

- Pour lire un ligne complète d'un fichier, utilisez **getline()** qui lit seulement une ligne à la fois.
- Exploitez un **input string stream (<sstream>)** pour analyser la ligne lue avec getline.

1.3.2) Ajoutez un troisième paramètre sur la ligne de commande qui permet d'indiquer de n'extraire que 2 coordonnées des sommets 3D. Si ce paramètre vaut:

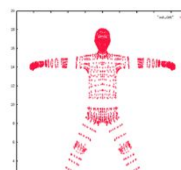
- 0: alors on extrait uniquement les coordonnées **x et y** des sommets 3D.
- 1: alors on extrait uniquement les coordonnées **y et z** des sommets 3D.
- 2: alors on extrait uniquement les coordonnées **z et x** des sommets 3D.

Attention: Respectez l'ordre d'extraction. Si le 3e paramètre vaut 2, alors on écrit d'abord la coordonnée Z puis la X.

Vérifiez vos résultats en procédant de la manière suivante:

- Dans un terminal, lancez **gnuplot**.
- Si le fichier à afficher s'appelle **out.dat**, tapez **plot 'out.dat'** puis enter

Exemple avec un exécutable s'appelant **parser_obj** :
%> ./parser_obj human.obj out.dat 0
%> gnuplot
gnuplot> plot 'out.dat'



Seconde partie : Type concret et Test unitaire d'un module

Buts

Dans cette partie, vous allez pratiquer une approche de mise au point d'un module exportant un type concret et une interface permettant de manipuler des variables de ce type.

Exercice 2.1 (niveau 0): usage de directives pour gérer l'inclusion multiple des fichiers en-tête (.h)

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

On appelle **type concret** un type dont on peut calculer l'espace mémoire nécessaire pour déclarer une variable de ce type. Sans l'information de l'espace mémoire nécessaire le compilateur n'accepte pas de déclaration de variable de ce type et produit une erreur de compilation. Tous les types de base sont des types concrets. Nous nous intéressons ici aux types que nous construisons pour les besoins d'une application, par exemple avec des structures.

On se propose de créer et de tester le module **etudiant** constitué de son *interface* **etudiant.h** et de son *implémentation* **etudiant.cc**.

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

Réponse : le moins possible. L'idée est de réduire autant que possible la dépendance entre le module **etudiant** et les autres modules qui vont inclure son interface **etudiant.h**.

Comme indiqué plus haut, le module **etudiant** est supposé fournir un type concret que nous allons appeler `Etudiant`. Tout module qui inclut **etudiant.h** doit ensuite pouvoir déclarer des variables de type Etudiant, comme par exemple :

```
Etudiant x,y ;
```

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

Deuxième conséquence : **etudiant.h** doit être inclus dans **etudiant.cc.** Pourquoi ?

1) *interface* **etudiant.h** :
1.1) d'abord adopter l'organisation à l'aide de directives qui permet d'éviter l'inclusion multiple (voir exercice 1 niveau 0)
1.2) Définir un modèle de structure `Etudiant` qui contient les champs nécessaires pour mémoriser:

- un nom de famille
- un prénom
- un age
- un code de 2 lettres pour une section

1.3) ajouter ensuite l'ensemble de déclarations de fonctions suivantes :

- une fonction qui initialise une variable de type `Etudiant` avec un dialogue utilisateur, avec vérification de la longueur maximum du champ section. La fonction renvoie la valeur de la structure.
- une fonction qui renvoie VRAI si deux variables de type `Etudiant` sont égales
- une fonction qui affiche dans le terminal sur une ligne les données d'une variable de type `Etudiant`

2) *implémentation* **etudiant.cc** : définir les fonctions déclarées dans *l'interface*. Ensuite compiler séparément le module **etudiant** de façon à produire un fichier **etudiant.o**.

3) test unitaire du module **etudiant**: écrire un module **test** (limité à une implémentation **test.cc**) qui exploite les trois fonctions exportées dans **etudiant.h**

Dans ce module **test** une fonction **main()** doit vérifier que les fonctions exportées par **etudiant.h** font bien ce qu'elles sont supposées faire. Pour des fonctions de type mathématique, il suffit d'appeler chaque fonction avec des valeurs de paramètres dont on connaît le résultat attendu. Si la fonction fournit le résultat attendu pour ces paramètres alors ce test est validé. A*ttention* : la validation d'un test ne veut pas dire que la fonction est validée pour n'importe quelles valeurs des paramètres ! Il faut donc prévoir un nombre suffisant de tests qui couvrent les cas de figure importants d'usage de la fonction.

Dans le cas de notre exercice, comme il ne s'agit pas de calculs mathématiques, on se propose de construire un vector de N structures étudiants, N étant la première donnée demandée par cette fonction, puis la fonction initialise cet ensemble étudiant par étudiant en vérifiant qu'il n'y a pas de doublon, et finalement affiche tous les étudiants contenus dans le tableau.

Ecrire le fichier Makefile qui compile séparément ce module **test**, le module **etudiant** et qui produit l'exécutable **test**. Exécuter en fournissant des valeurs au clavier.

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)** ?

5) Remarque finale : un type concret est facile à mettre en œuvre mais présente une faiblesse car le module responsable du type (ici le module **etudiant**) ne peut pas empêcher un module indépendant (ici le module **test**) de modifier les champs de la variable `Etudiant` car ce module a accès au modèle de structure `Etudiant`.

C'est une perte de contrôle qui fait que le module **etudiant** ne peut pas être tenu responsable d'erreurs qui pourraient se produire à cause de valeurs incorrectes des champs d'une variable de type Etudiant.

C'est pourquoi nous utilisons des classes pour les types les plus élaborés du projet.