TP C avancé n^o 5
G. Subrenat, Y. Bertrand, N. Courilleau, D. Meneveaux

Licence Informatique L3 - 2020-2021
UFR SFA - Université de Poitiers

Reprise en main

Table des matières

1 Résumé et cadre de travail

Dans ce TP il s'agit de maîtriser la notion d'abstraction pointeur pour masquer les données d'une "classe" (module étant un terme plus approprié en C).

L'abstraction pointeur permet de masquer les données d'un module dans le .c pour interdire une manipulation directe dans les autres .c

2 Explications générales

Le but est de manipuler une collection ordonnée de voitures.

2.1 Gestion des erreurs

Les erreurs (indice incorrect dans un tableau, kilométrage négatif, erreur d'allocation, ...) seront gérées de manière brutale avec des *assert*.

Le module *myassert* est fourni : il fonctionne de la même façon que le module système *assert* mais est plus verbeux et permet des affichages personnalisés.

Dans ce module, seule la macro *myassart* doit être utilisée (cf. fichier *myassert.h* pour son utilisation). Mais vous êtes encouragés à regarder le code complet des fichiers *myassert.h* et *myassert.c*.

Il y a deux types d'erreur :

- les bugs de programmation : l'utilisation des *assert* n'est pas choquante car les bugs doivent être corrigés avant qu'un logiciel ne soit mis en production.
- les erreurs imprévisibles comme un manque de mémoire : il faudrait alors un mécanisme de gestion plus subtil comme les exceptions pour permettre au programme de gérer l'erreur.

Dans notre cas, par soucis de facilité [1], nous n'utiliserons que des assertions.

On demande de gérer un maximum d'erreurs : il y aura fréquemment deux ou trois *assert* en débuts de fonctions.

2.2 Compilation

Il est indispendable de compiler (et tester lorsque c'est possible) le code au fur et à mesure de la programmation.

Un Makefile est fourni pour automatiser et optimiser le processus de compilation.

L'utilisation, en ligne de commandes, suit le schéma suivant :

```
$ make <cible>
```

où `<cible>` est le fichier que l'on veut créer. S'il n'y a pas de cible précisée, alors c'est la première cible décrite dans le fichier *Makefile* qui est construite, généralement l'exécutable final.

Par exemple, si on veut compiler uniquement le fichier *Voiture.c* pour obtenir le fichier *Voiture.o* :

```
$ make Voiture.o
```

Et pour obtenir l'exécutable final (qui s'appelle *main* dans notre cas) :

```
$ make main
```

où plus simplement :

```
$ make
```

1. sans compter que les exceptions en C sont un vrai cauchemar

Vous pouvez étudier la construction du *Makefile*, mais ce n'est pas au programme de cette unité d'enseignement.

2.3 Architecture logicielle

Il y aura 3 modules (en plus de *myassert*) :
- *Voiture.h*/ *Voiture.c* : gestion d'une seule voiture
- *Collection.h*/ *Collection.c* : gestion d'une collection ordonnée de voitures
- *main.c* : le programme principal

Pour insister lourdement :
- Les données du module *Voiture* sont définies dans *Voiture.c*
- *main.c* et *Collection.c* ne peuvent pas manipuler directement la structure de *Voiture* et devront passer par les méthodes de *Voiture*.

3 Débugage

3.1 Debugueur classique

Pour utiliser un débugueur classique, il faut ajouter l'option *-g* lors de la compilation (dans le *Makefile* fourni cette option est activée).

Le debugueur classique permet d'exécuter un programme ligne par ligne et d'afficher (et modifier) les variables.

Par exemple, *kdbg* est un debugueur épuré et assez intuitif. Il se lance de la façon suivante :

```
$ kdbg <nom exécutable>
```

Voici une aide assez succincte :
- pour ajouter/supprimer un point d'arrêt, il suffit de cliquer à gauche du numéro de ligne (et à gauche du petit signe plus)
- pour lancer l'exécution : bouton “Exécuter” ou touche *F5*
- pour exécuter une ligne : bouton “Prochaine instruction” ou touche *F10*
- pour entrer dans une fonction : bouton “Entrer dans cette fonction” ou touche *F8*
- mettre la souris sur une variable affiche sa valeur
- la partie en haut à droite affiche les variables locales de la fonction courante
- la partie en bas à gauche affiche la pile d'appels et il est possible de naviguer dedans

3.2 Debugueur mémoire

Pour utiliser un débugueur mémoire, il est préférable d'ajouter l'option *-g* lors de la compilation.

Le débugueur mémoire permet de détecter les accès mémoire erronés :
- écriture hors d'une zone allouée
- lecture hors d'une zone allouée
- lecture d'une zone mémoire autorisée mais non initialisée
- double libération d'une zone mémoire

Il permet également de détecter les fuites mémoire.

Nous utilisons *valgrind* dans ce TP.

Voici la syntaxe de *valgrind* :

```
$ valgrind [options valgrind] <nom exécutable> [paramètres exécutable]
```

Pour lancer simplement *valgrind* sur l'exécutable généré par le Makefile :

```
$ valgrind ./main
```

ou

```
$ valgrind ./main full
```

Pour avoir des informations sur les fuites mémoire :

```
$ valgrind --leak-check=full ./main full
```

Lors d'une écriture (ou lecture) interdite, *valgrind* précise :
- le fichier, la fonction et la ligne où l'erreur s'est produite
- le fichier, la fonction et la ligne où l'allocation s'est effectuée
- combien d'octets après la zone allouée l'accès a eu lieu

Lors de fuites mémoire, *valgrind* précise :
- le nombre de blocs alloués, le nombre de blocs désalloués
- la quantité de mémoire perdue
- pour chaque erreur : le fichier, la fonction et la ligne où l'allocation s'est effectuée

Dans ce TP, vous devez systématiquement lancer votre exécutable avec *valgrind* qui ne devra relever aucune erreur.

4 Spécifications

4.1 Spécification du module Voiture

Une voiture contient :
- sa marque : une chaîne de caractères
- son année de fabrication : un entier
- son kilométrage : un entier
- la liste des diverses immatriculations au cours de sa vie : une suite ordonnée de chaînes de caractères

Les opérations possibles sur une voiture sont :
- créer une voiture : les quatre caractéristiques sont passées en paramètres
- créer une voiture : les caractéristiques sont récupérées dans une voiture déjà existante (c'est donc une copie de voiture)
- créer une voiture : les caractéristiques sont récupérées dans un fichier (fichier écrit par l'avant-dernière fonction de cette liste)
- échanger les données de deux voitures (permutation des valeurs)
- détruire une voiture
- les accesseurs (en lecture et/ou écriture) sur les différents champs :
  . lire la marque (c'est une copie de la chaîne que l'on récupère)
  . il est impossible de modifier la marque (i.e. lecture seule)
  . lire l'année de fabrication
  . il n'est pas possible de modifier l'année de fabrication (i.e. lecture seule)
  . lire le kilométrage
  . modifier le kilométrage, mais le nouveau kilométrage ne peut être que plus grand que l'ancien
  . lire le nombre d'immatriculations
  . récupérer une immatriculation via sa position dans la suite (c'est une copie de la chaîne que l'on récupère)
  . ajouter une immatriculation
- afficher une voiture à l'écran

- sauvegarder une voiture dans un fichier : le paramètre est un FILE * et non un nom de fichier
- réinitialiser une voiture en récupérant les données dans un fichier. Le fichier aura l'organisation induite par la méthode précédente. De même c'est un FILE * qui est passé en paramètre. Ce n'est pas un constructeur : c'est une voiture déjà créée qui est passée en paramètre.

En outre on veut comptabiliser :
- le nombre de voitures initialisées (par les trois premières fonctions)
- le nombre d'immatriculations créées (par les trois premières fonctions, et par la méthode d'ajout d'une immatriculation)
- la plus petite année de fabrication rencontrée
- la plus grande année de fabrication rencontrée

C'est à la charge du module de mettre à jour ces données de manière transparente pour l'utilisateur ; mais ce dernier a des méthodes d'accès en lecture à ces données.

4.2 Spécification du module Collection

Une collection contient :
- une suite (ordonnée) de Voitures
- une information indiquant si le tableau est trié (par dates croissantes) ou non

Les opérations possibles sur une collection sont :
- créer une collection vide
- créer une collection en faisant une copie d'une collection existante
- détruire une collection (qui ne pas être réutilisée)
- vider une collection (qui peut être réutilisée ultérieurement)
- lire la taille de la collection
- récupérer une voiture selon sa position dans la suite (c'est une copie de la voiture que l'on récupère)
- ajouter une voiture sans se préoccuper de conserver le tri s'il est effectif (l'idée étant de sacrifier le tri pour une efficacité accrue d'insertion) ; c'est une copie de la voiture qui est insérée dans la collection
- ajouter une voiture en conservant le tri ; et si la suite n'est pas triée lors de l'appel de la méthode, c'est une erreur. C'est une copie de la voiture qui est insérée dans la collection.
- supprimer une voiture par sa position sans se préoccuper de conserver le tri s'il est effectif (on privilégie à nouveau l'efficacité)
- supprimer une voiture par sa position en conservant le tri ; c'est à nouveau une erreur si la méthode est appelée sur une suite non triée
- trier la suite
- afficher la collection entière à l'écran
- sauvegarder toute la collection dans un FILE *
- réinitialiser la collection en récupérant les données dans un FILE *. Le fichier aura l'organisation induite par la méthode précédente. Ce n'est pas un constructeur : c'est une collection déjà créée qui est passée en paramètre.

Comme cela a été annoncé, une collection stocke des copies des voitures qu'elle reçoit, que ce soit le constructeur par copie ou les méthodes d'ajout d'une voiture.
C'est donc à la collection de détruire ses voitures lorsqu'elle n'en a plus besoin.

4.3 Spécification du module main

Il ne sert qu'à tester les deux modules précédents.

Lors du lancement dans le shell, l'exécutable prend un paramètre facultatif. S'il est fourni, il vaut obligatoirement "full".

S'il n'y a pas de paramètre, on teste toutes les fonctions sauf celles qui manipulent les fichiers. S'il y a un paramètre, on teste en plus les fonctions qui manipulent les fichiers.

5 Exercice 1

Le but est de montrer que l'ont peut écrire un programme (*main.c*) utilisant des bibliothèques (*Voiture* et *Collection*) sans connaître leurs implémentations [2] et même sans que ces dernières ne soient écrites. En revanche il faut connaître leurs interfaces (i.e. les *.h*).

Écrire uniquement les fichiers suivants (à l'exclusion de tout autre) :
- *Voiture.h* entièrement (avec l'abstraction pointeur)
- *Collection.h* entièrement (avec l'abstraction pointeur)
- *main.c*

La fonction *main* devrait contenir des appels à toutes les méthodes déclarées dans les .h pour faire une couverture de tests complète. Dans cet exercice il vous est seulement demandé de choisir 5 méthodes de chaque module et des les appeler.

Le fichier *main.c* doit compiler sans erreur ni warning :

```
$ make main.o
```

ou

```
$ gcc -g -Wall -Wextra -pedantic -std=c99 -c main.c
```

Comme les deux modules ne sont pas implémentés, il possible de compiler le fichier *main.c*, mais pas de générer l'exécutable (édition de liens).

Sont fournis les fichiers suivants :
- *Makefile*
- *myassert.h*

Le fichier *myassert.h* n'est *a priori* pas à utiliser dans cette partie

6 Exercice 2

On oublie les 3 fichiers précédents, de nouvelles versions sont fournies et imposées (i.e. obligation de les utiliser et interdiction de les modifier par la suite).

En outre les versions compilées de *Voiture.c* et *Collection.c* sont également fournies (en espérant qu'elles sont compatibles avec votre architecture).

Pour récapituler, voici la liste des fichiers disponibles :
- *Makefile*
- *myassert.h*
- *myassert.c*
- *main.c*
- *Voiture.h*
- *Voiture.o*
- *Collection.h*
- *Collection.o*

Le seul et unique but de cet exercice est de vérifier que l'ensemble compile :

```
$ make
```

ou

```
$ gcc -g -Wall -Wextra -pedantic -std=c99 myassert.c main.c Collection.o Voiture.o -o main
```

2. D'ailleurs lorsqu'on utilise la bibliothèque mathématique (*math.h*), s'intéresse-t-on au codage de la fonction *sqrt* ?

et de lancer l'exécution :

```
$ ./main
```

Si vous passez plus de 60 secondes sur cet exercice, cela signifie que vous faites plus que ce qui est demandé (comme écrire du code par exemple).

7 Exercice 3

Écrire uniquement le fichier *Collection.c* pour générer votre propre fichier *Collection.o*.

Comme le module *Collection* utilise *Voiture*, le fichier *Voiture.o* (i.e. la version compilée de *Voiture.c*) est fourni.

Les voitures sont stockées dans un tableau alloué dynamiquement (on impose que ce soit un tableau). Le nombre de cases du tableau correspond exactement au nombre de voitures stockées.

Compiler avec le *main.c* et le *Voiture.o* fournis, et vérifier que tout fonctionne :

```
$ make
```

ou

```
$ gcc -g -Wall -Wextra -pedantic -std=c99 myassert.c main.c Collection.c Voiture.o -o main
```

Rappel : interdit de modifier *main.c*, *Collection.h* ou *Voiture.h*. Si vous êtes tentés de les modifier, alors vous ne répondez pas à la question.

Rappel : *Collection.c* n'a pas le droit de manipuler directement les champs de la structure Voiture, mais juste le droit d'appeler les méthodes de *Voiture.h* (et si *Collection.c* arrive à manipuler directement la structure de Voiture je suis intéressé de savoir comment vous avez fait).

Sont fournis les fichiers suivants :

- *Makefile*
- *myassert.h*
- *myassert.c*
- *main.c*
- *Voiture.h*
- *Voiture.o*
- *Collection.h*

8 Exercice 4

Écrire le fichier *Voiture.c* pour générer votre propre fichier *Voiture.o*

Les chaînes de caractères sont allouées dynamiquement et aux tailles strictement nécessaires (pas de surdimensionnement).
Il en va de même pour le tableau des immatriculations (on impose que les immatriculations soient stockées dans un tableau).

Testez votre code avec les deux compilations suivantes :

- avec le fichier *Collection.o* fourni et votre *Voiture.c* (pour se concentrer sur le débogage du module *Voiture*)
- avec vos deux fichiers *Voiture.c* et *Collection.c*

Rappel : dans cet exercice, seul le fichier *Voiture.c* doit être écrit et il est interdit de modifier *main.c*, *Collection.h* ou *Voiture.h*

Sont fournis les fichiers suivants :

- *Makefile*
- *myassert.h*
- *myassert.c*
- *main.c*
- *Voiture.h*
- *Collection.h*
- *Collection.o*