Devoir noté – BDC : une petite BD en C

Merlin Nimier-David Jean-Cédric Chappelier

du 9 mars 2020, 12h00, au 23 mars, 23h59.

I. Introduction et instructions générales

Ce devoir noté consiste à écrire une petite application de gestion de base de données, avec des fonctionnalités incluant : chargement de la base depuis un fichier texte, interrogation de la base selon plusieurs critères, et écriture des résultats dans divers fichiers texte.

Nous vous demandons d'écrire tout votre code dans *un seul* fichier nommé `db.c` sur la base d'un fichier fourni, à compléter.

Indications : Si un comportement ou une situation donnée n'est pas définie dans la consigne ci-dessous, vous êtes libre de définir le comportement adéquat. On considérera comme comportement adéquat toute solution qui ne viole pas les contraintes données et qui ne résulte pas en un crash du programme.

Instructions :

1. Cet exercice doit être réalisé **individuellement !** L'échange de code relatif à cet exercice noté est **strictement interdit !** Cela inclut la diffusion de code sur le forum ou sur tout site de partage.

   Le code rendu doit être le résultat de *votre propre production*. Le plagiat de code, de quelque façon que de soit et quelle qu'en soit la source sera considéré comme de la tricherie (et peut même, suivant le cas, être illégal et passible de poursuites pénales).

   En cas de tricherie, vous recevrez la note « NA » pour l'entièreté de la branche et serez de plus dénoncés et punis suivant l'ordonnance sur la discipline.

2. Vous devez donc également garder le code produit pour cet exercice dans un endroit à l'accès strictement personnel.

   Le fichier (source !) `db.c` à fournir comme rendu de ce devoir ne devra plus être modifié après la date et heure limite de rendu.

3. Veillez à rendre du code *anonyme* : **pas** de nom, ni de numéro SCIPER, ni aucun identifiant personnel d'aucune sorte !

4. Utilisez le site Moodle du cours pour rendre votre exercice.

   Vous avez jusqu'au lundi 23 mars, 23:59 (heure du site Moodle du cours faisant foi) pour soumettre votre rendu. **Aucun délai supplémentaire ne sera accordé.**

Je tiens également à rappeler ici (dit dans le premier cours) que l'objectif de ce devoir n'est pas tant de vous évaluer que de vous entraîner. C'est pour cela que le coefficient de ce devoir est assez faible (1 sur 10). C'est également dans cet esprit qu'il a été conçu : de façon très progressive (de plus en plus difficile). Utilisez le donc comme un entraînement indicatif (sur lequel vous aurez un retour) en travaillant à votre niveau dans une durée qui vous semble raisonnable pour les crédits de ce cours (au lieu d'essayer de tout faire dans un temps déraisonnable). Ce coefficient (1 sur 10) n'est pas représentatif de la charge de travail (qui peut beaucoup varier à ce niveau d'un(e) étudiant(e) à l'autre), mais bien pour vous donner l'occasion d'avoir un premier retour sur votre niveau de programmation en C, sans prendre trop de « risques ».

II. Contexte

Les Bases de Données (BdD) font partie des programmes les plus importants et les plus utilisés au monde. Elles sont utilisées pour stocker tout type d'informations, allant des comptes bancaires aux informations médicales. Ici, nous vous proposons d'implémenter une version très simplifiée d'un programme de gestion de BdD permettant de stocker les informations d'élèves participants au cours.

Limitations : étant donné que les allocations dynamiques n'ont pas encore été vues en cours au moment où ce sujet est donné, nous nous limiterons à des bases de données dont la taille maximale est petite et connue (allocation statique). De même, le nombre d'éléments retournés par les requêtes ne devra pas dépasser une taille connue et fixée. Cela nous permet de stocker les éléments et résultats dans des tableaux de taille fixe. En pratique, toutes les BdD existantes peuvent gérer un nombre dynamique d'éléments, mais cela n'est pas du tout demandé dans ce devoir.

III. Implémentation

Le fichier `db.c` distribué contient un certain nombre de fonctions, dont certaines que nous vous demandons de compléter. (Commencez peut être par jeter juste un coup d'oeil au fichier fourni pour avoir une idée.)

III.1 Types

Le programme sera amené à manipuler un petit nombres de concepts, que nous vous proposons de représenter par des types et structures de données que les fonctions pourront ensuite utiliser. Il est important de choisir ces types avec

attention, puisqu'ils pourraient rendre l'écriture des fonctions plus difficile que nécessaire.

Notez que les fonctions fournies utilisent déjà les types que vous allez définir : veillez donc à ce que ces fonctions compilent et fonctionnent comme prévu, sans les modifier.

Catégorie d'étudiants

Définissez le type `StudentKind` comme une énumération permettant de distinguer entre trois catégories d'étudiant(e)s : `Bachelor`, `Master` et `Exchange`.

Ajoutez également la valeur `StudentKindCount` correspondant au nombre d'entrées de l'énumération.

SCIPER

Utilisez un simple `typedef` pour assurer que le type `SCIPER` utilisé pour représenter le SCIPER des étudiant(e)s (nombre entier d'identification unique) sera codé sur au moins 20 bits.

Note : dans cet exercice, il n'est pas demandé de trouver ou créer un type *parfaitement adapté* au stockage du SCIPER ; on se contentera d'utiliser le type entier existant le plus proche.

Etudiant(e)s

En utilisant les types créés ci-dessus, définissez le Type `Student` comme une structure de données contenant les informations suivantes :

- un numéro SCIPER ;
- trois réels `grade_sn`, `grade_hw`, et `grade_exam` pour stocker trois notes ;
- un champ pour identifier la catégorie de l'étudiant(e) ;
- un pointeur `teammate` vers un(e) autre étudiant(e), permettant de représenter la mise en binôme de deux étudiant(e)s (expliquée plus loin).

BdD

Le type `Database` est le type principal manipulé par ce programme. Dans cet exercice, la BdD est simplement représentée par un tableau d'étudiant(e)s, dont la taille est connue et fixée, donnée par `DB_MAX_SIZE` (fourni).

Requêtes

Un certain nombre des fonctions à implémenter (voir plus bas) consistent à trouver et retourner un sous-ensemble des étudiant(e)s stocké(e)s dans la BdD selon différents critères. Par souci d'efficacité, nous **ne** souhaitons **pas** *copier* toutes les informations relatives aux étudiant(e)s sélectionné(e)s, mais plutôt y accéder « par référence ».

On utilisera pour cela le type `QueryResult` qui doit être un tableau de taille fixe de pointeurs vers des étudiant(e)s, dont la taille est donnée par `QUERY_MAX_SIZE` (fourni).

III.2 Fonctions fournies

Avant de décrire plus bas le travail à faire, nous présentons ici les fonctions déjà fournies (pouvant être utiles).

Fonction `main()`

La fonction `main()` appelle les diverses fonctions (dont celles que vous aurez définies) afin de les tester. Tout d'abord, le fichier dont le nom est passé en argument au programme (ou bien `"data.txt"` par défaut) est ouvert afin de le charger en mémoire. Le contenu de la base est ensuite affiché, puis différentes requêtes sont effectuées. Les résultats de chaque requête sont stockés dans un fichier séparé.

Utilisez donc ce `main()` fourni pour faire tous les tests qui vous semblent utiles (typiquement en regardant les résultats produits dans des cas bien spécifiques que vous aurez jugé pertinent de choisir).

Pensez également à tester (par vous-mêmes) chacune de vos fonctions, progressivement.

Fonctions outils

Les fonctions outils suivantes vous sont fournies :

- `grade_average()` : pour un(e) étudiant(e) donné(e), calcule la moyenne pondérée de ses trois notes.
- `team_diff()`: pour un(e) étudiant(e) donné(e), calcule la différence absolue entre sa moyenne et celle de son/sa binôme.
- Fonctions `fprintf_student_kind()`, `write_student()` et `write_query_results()` :

  ces fonctions écrivent une représentation textuelle de différents types manipulés par le programme, vers un fichier ou le flux de sortie standard ;

  nous fournissons ces fonctions pour garantir que les résultat de vos programmes soient écrits dans un format consistant ; **il est donc important de ne pas les modifier** ;

  n'hésitez cependant pas à les utiliser lors de la rédaction de vos fonctions pour vérifier que tout est en ordre.

III.3 Fonctions à implémenter

Remarque préliminaire : dans ce devoir, il n'est pas demandé d'implémenter chaque fonction de manière algorithmiquement optimale. Grâce à la petite

taille des BdD traitées, une fonction de complexité même quadratique devrait tout de même terminer presque instantanément. (Mais ne faites pas pire que quadratique, tout de même ! `;-)`)

Conventions

Bien que la taille des tableaux de type `Database` et `QueryResult` soit fixée, ils ne seront pas toujours utilisés au maximum de leur capacité. Par exemple, la BdD pourrait être représentée par un tableau de taille 20, mais n'avoir que 6 étudiants qui y figurent.

Afin de limiter le nombre de paramètres à manipuler pour chaque fonction, nous adoptons les conventions suivantes (assez communes en C) :

- la première entrée d'un tableau `Database` dont le SCIPER est `0` marque la fin de la BdD ; si toutes les entrées ont un SCIPER strictement supérieur à zéro, alors la base est remplie (`DB_MAX_SIZE` entrées valides) ;
- de même, la première entrée d'un tableau `QueryResult` dont la valeur est `NULL` marque la fin des résultats ; si le premier pointeur de `QueryResult` est `NULL`, c'est qu'il n'y a aucun résultat ; si aucun pointeur dans le tableau ne vaut `NULL`, alors il y a `QUERY_MAX_SIZE` résultats valides ; on ne considérera *jamais* dans ce devoir le cas de requêtes ayant plus que `QUERY_MAX_SIZE` résultats valides.

Comme vous l'avez peut-être remarqué dans les entêtes des fonctions fournis, nous avons adopté une seconde convention assez courante concernant les types de retour : les fonctions étant susceptibles d'échouer, p.ex. `load_database()`, ne retournent pas directement leur résultat, mais retournent plutôt un *code d'erreur*. Par convention, la valeur de retour doit être égale à `0` si tout a fonctionné correctement, ou un nombre entier non nul dans le cas contraire. Bien que cela ne soit pas demandé dans ce devoir, les différents codes de retour pourraient être utilisés pour présenter des erreurs plus détaillées à l'utilisateur, permettant, p.ex., de gérer les différents cas de manière adaptée. Ce ne sera pas le cas ici.

La base de donnée elle-même est « retournée » en écrivant directement dans le tableau `db_out`.

Fonction `db_entry_count()`

La fonction `db_entry_count()` doit simplement suivre la convention expliquée ci-dessus pour calculer la taille d'une BdD fournie, c.-à-d. le nombre d'entrées valides.

Note : dans une application plus réaliste, on éviterait à tout prix de devoir parcourir toute la base pour déterminer le nombre d'entrées. Dans notre cas, le nombre maximum d'entrées est suffisamment petit pour que ça n'ait pas d'importance (et ça nous permet de vous demander de coder une fonction assez simple `;-)`).

Fonction `get_student_by_sciper()`

La fonction `get_student_by_sciper()` retourne un pointeur vers l'étudiant(e) ayant le SCIPER donné en argument, ou `NULL` si aucun étudiant ne correspond.

Fonction `load_database()`

La fonction `load_database()` est responsable du chargement en mémoire des entrées de la BdD depuis un fichier texte. Le nom du fichier à ouvrir est passé en argument. Le fichier contiendra toujours une liste d'étudiants séparés par une ligne vide, avec les informations suivantes (une par ligne) :

```
SCIPER (entier)
grade_sn (réel)
grade_hw (réel)
grade_exam (réel)
type d'étudiant (entier)
SCIPER du teammate (entier)

SCIPER de l'étudiant suivant (entier)
[etc]
```

Vous pouvez consulter le fichier `data.txt` fourni pour exemple. Les entrées doivent être chargées dans la base dans le même ordre que le fichier.

Note : le format pour lire un SCIPER est donné en commentaire ; remplissez les deux autres champs de `fscanf()`, mais ne modifiez pas le champs format.

Les erreurs suivantes doivent être détectées lorsque le fichier est parcouru :

- le fichier donné ne peut pas être ouvert en lecture ;
- un champ attendu ne peut pas être lu ;
- le type de l'étudiant ne fait pas partie des trois types définis par l'énumération `StudentKind`.

En cas d' erreur, la fonction affichera un message pertinent et retournera `-1`. Le contenu de `db_out` n'est pas spécifié dans ce cas (« *undefined behavior* »).

Après avoir chargé les données en mémoire, la fonction doit :

1. s'assurer que la convention indiquant la fin de la BdD est respectée, en mettant le SCIPER de la première entrée invalide à `0` ; (**Note :** cela peut aussi avoir été fait avant. . .) ;
2. utiliser le numéro de SCIPER indiqué pour le binôme de chaque étudiant afin de faire pointer le champ `teammate` de cet(te) étudiant(e) vers l'autre étudiant(e) correspondant(e) ; un SCIPER égal à `0` est légal ici, et le pointeur `NULL` doit dans ce cas être utilisé (cela indique un(e) étudiant(e) sans binôme) ;

   indice : pensez à la fonction `get_student_by_sciper()` ;

3. vérifier que les binômes sont consistants en utilisant la fonction `check_teammates_consistency()` (voir ci-dessous) ; toute inconsistance détectée est considérée comme une erreur, et la fonction doit alors retourner `-1`.

Fonction `get_students_by_type()`

La fonction `get_students_by_type()` retourne au plus `QUERY_MAX_SIZE` pointeurs vers les étudiants dont le type correspond au type recherché (passé en argument), dans l'ordre où ils sont stockés dans la BdD.

Fonction `check_teammates_consistency()`

Pour une BdD donnée, cette fonction doit vérifier la consistance des binômes définis par le champ `teammate` de chaque étudiant. En particulier, seuls les cas suivants sont acceptés pour un(e) étudiant(e) donné(e) :

- il/elle n'a pas de binôme, son champ `teammate` est alors `NULL` ;
- il/elle a un binôme *valide*: son champ `teammate` pointe vers un(e) autre étudiant(e) existant de la BdD dont le champ `teammate` pointe vers lui/elle : le `teammate` de A pointe vers B et le `teammate` de B pointe vers A.

La fonction doit retourner `0` si tout est en ordre, ou `-1` en cas d'inconsistance.

En cas d'erreur, cette fonction devra de plus indiquer clairement (par un message) la source d'erreur ; p.ex. :

```
SCIPER1 a SCIPER2 comme binome mais SCIPER2 n'a pas de binome
```

ou

```
SCIPER1 a SCIPER2 comme binome mais SCIPER2 a SCIPER3 comme binome
```

Fonction `get_least_homegenous_teams()`

Note : cette fonction est significativement plus difficile à implémenter. Comme dit en préambule, consacrez-y le temps que vous estimez raisonnable.

La fonction `get_least_homegenous_teams()` doit retourner les `QUERY_MAX_SIZE` binômes ayant les écarts de moyenne les plus importants entre ses deux membres. Le tableau `result_out` devra contenir un pointeur par binôme (sans duplication de binôme), pointant vers l'étudiant(e) ayant la moyenne la plus faible des deux membres du binôme retenu. (Relisez plusieurs fois le paragraphe précédent, faites des schémas, regardez l'exemple ci-dessous.)

Utilisez les fonctions `grade_average()` et `team_diff()` fournies, ainsi que la macro `student_to_index()` qui permet de calculer l'indice d'un(e) étudiant(e) dans une BdD `tab` à partir de son adresse (= à partir d'un pointeur sur lui/elle). Les macros et surtout l'arithmétique des pointeurs seront vues plus tard dans le cours ; il n'est donc pas attendu que vous compreniez cette macro. Utilisez la simplement si nécessaire.

Remarque : si la BdD possède moins de `QUERY_MAX_SIZE` binômes en tout, alors, par définition, on s'attend à ce que `result_out` contienne un pointeur vers l'étudiant(e) ayant la moyenne la plus faible de chaque binôme de la base.

Exemple : le fichier `many_teams.txt` fourni contient 20 étudiantes (numérotées de 1 à 20), dont 18 sont en binômes (les étudiantes 6 et 12 ne le sont pas).

Puisque `QUERY_MAX_SIZE` est fixé à 5, on aura au début de l'algorithme (jusqu'à l'étudiante 8) les cinq premiers binômes dans le résultat de la requête, mais pointant sur les étudiantes 20, 2, 3, 7 et 8 :

```
0000020 - 4.50, 4.75, 5.00 - bachelor - 000001
0000002 - 3.25, 2.75, 5.25 - master   - 000018
0000003 - 1.50, 0.50, 6.00 - master   - 000004
0000007 - 5.50, 5.75, 5.75 - bachelor - 000005
0000008 - 5.50, 5.50, 5.50 - bachelor - 000011
```

puisqu'en effet dans chacun cinq premiers des binômes ce sont elles qui ont les plus petites moyenne (p.ex. la moyenne de l'étudiante 20 est de 4.775, alors que celle de l'étudiante 1 est de 4.95). A noter ici que le binôme (3, 4), ni le binôme (5, 7) ne sont représentés deux fois.

Le plus petit des écarts entre binômes à ce stade est de 0.075 pour le binôme (5,7). Le prochain binôme considéré (9, 11) ayant un écart de moyenne entre ses deux membres de 0.05, il n'entre pas dans le résultat de la requête.

Le binôme considéré ensuite est (13, 15), dont l'écart de moyenne est de 3.35. Ce binôme prend donc la place du binôme (5, 7). Et comme l'étudiant 13 à une moyenne plus faible que l'étudiante 15, c'est elle qui entre dans le tableau des résultat. On a donc à ce stade :

```
0000020 - 4.50, 4.75, 5.00 - bachelor - 000001
0000002 - 3.25, 2.75, 5.25 - master   - 000018
0000003 - 1.50, 0.50, 6.00 - master   - 000004
0000013 - 1.50, 2.75, 3.00 - bachelor - 000015
0000008 - 5.50, 5.50, 5.50 - bachelor - 000011
```

dont le plus petit écart de note est maintenant 0.175 (binôme (1, 20)).

On procède ainsi de suite jusqu'au bout pour arriver au final au résultat :

```
0000016 - 4.50, 4.75, 4.00 - bachelor - 000014
0000002 - 3.25, 2.75, 5.25 - master   - 000018
0000003 - 1.50, 0.50, 6.00 - master   - 000004
0000013 - 1.50, 2.75, 3.00 - bachelor - 000015
0000008 - 5.50, 5.50, 5.50 - bachelor - 000011
```

IV. Quelques conseils pour terminer

N'hésitez pas à créer d'autres fonctions utilitaires si nécessaire.

Tout votre code et toutes vos fonctions doivent être robustes tant que faire se peut.

Pensez à tester correctement chacune des fonctionnalités implémentées à **chaque** étape et **avant** de passer à l'étape suivante. Cela vous évitera bien des tracas.