MM. Clément KOPERSKI et Sami AGGAR, élèves en Systèmes Embarqués 3, vous présentent :

Rapport du Projet de Programmation Avancée

–

Application d'analyse de 58492 vols aux États-Unis en 2014

INTRODUCTION

Ce projet de Programmation Avancée, qui vient clôre le module, a pour but de charger 3 fichiers au format CSV (*comma-separated values*) dans des structures de données adéquates, puis de coder les requêtes que l'utilisateur pourra renseigner via le terminal, afin d'interroger la base de données.

La base de données sera constituée des 58492 vols d'avions ayant eu lieu à l'intérieur du territoire des États-Unis en 2014. Ainsi, les 3 fichiers CSV que nous avons à disposition sont :

- "airlines.csv" : Tableau des compagnies aériennes américaines.
- "airports.csv" : Tableau des aéroports américains.
- "flights.csv" : Tableau des vols ayant eu lieu en 2014 à l'intérieur des États-Unis.

Le cahier des charges est le suivant :

1. Charger les données des 3 fichiers CSV dans des structures de données jugées pertinentes.
2. Attendre une commande renseignée par l'utilisateur.
3. Traiter la commande.
4. Afficher le résultat de cette commande.
5. Revenir à l'étape 2.

Au cours de ce projet, **nous avons fait des choix** concernant les structures de données utilisées pour stocker les vols, aéroports & compagnies aériennes que nous allons expliquer par la suite. Également, nous avons codé les requêtes, le menu et l'affichage d'une certaine manière, tout en étant conforme au cahier des charges. Enfin, nous expliquerons le fonctionnement de notre programme et ses limites.

Nous précisons que tous nos codes ont été commentés de manière à ce qu'ils soient facilement compréhensibles à leur lecture. Nous indiquons toutefois que nous avons produit du **code en langage C** qui est loin d'être parfait, mais qui est fonctionnel et suffisamment lisible pour ce projet.

I - Nos choix de structures de données

Tout d'abord, nous avons analysé les données brutes dans les fichiers CSV. Nous souhaitions connaître le volume d'informations à traiter, repérer de potentielles clés intéressantes pour réaliser des tables de hachage (notamment pour trier les nombreux vols) et repérer de potentielles particularités suite aux tests de nos structures de données. Voici ce que nous avons relevé d'intéressant :

- **Dans le fichier "airlines.csv" :** Le plus simple des 3 fichiers de données, car il ne contient que 2 colonnes ("IATA_CODE" & "AIRLINE") & 15 lignes → **Très faible volume d'informations** à traiter. Il n'y a en effet que 14 compagnies aériennes recensées aux États-Unis en 2014.
  De plus, **le fichier est complet** (aucune donnée manquante).

- **Dans le fichier "airports.csv" : Volume de données intermédiaire**, car 7 colonnes ("IATA_CODE", "AIRPORT", "CITY", "STATE", "COUNTRY", "LATITUDE" & "LONGITUDE") & 323 lignes. Il y a en effet 322 aéroports recensés aux États-Unis en 2014.
  De plus, **le fichier est quasi complet** (il manque les données pour les champs "LATITUDE" & "LONGITUDE" des aéroports lignes 98, 236 & 315).

- **Dans le fichier "flights.csv" : Volume de données conséquent**, car 14 colonnes ("MONTH", "DAY", "WEEKDAY", "AIRLINE", "ORG_AIR", "DEST_AIR", "SCHED_DEP", "DEP_DELAY", "AIR_TIME", "DIST", "SCHED_ARR", "ARR_DELAY", "DIVERTED", "CANCELLED") & 58493 lignes. Il y a en effet 58492 vols recensés aux États-Unis en 2014.
  De plus, **le fichier n'est pas complètement renseigné** (il y a 2869 champs vides, correspondant majoritairement aux vols annulés, qui ne peuvent donc pas disposer des informations concernant l'heure de départ, d'arrivée, etc…).

Ensuite, nous avons rapidement choisi de stocker les données concernant les compagnies aériennes (fichier "airlines.csv") et les aéroports (fichier "airports.csv") dans des listes chaînées, car le volume de données est faible et nous souhaitons également que l'ajout et la suppression soient dynamiques. Ainsi, les listes chaînées permettent (dans la limite de la mémoire RAM du PC), d'allouer et de désallouer dynamiquement des emplacements mémoires. Ce qui permet de stocker plus ou moins de compagnies aériennes & d'aéroports, sans limite prédéfinie. Ainsi, si notre programme venait à être

réutilisé pour stocker les compagnies aériennes & les aéroports des États-Unis en 2021, alors même qu'il y a eu 5 compagnies aériennes de plus et 50 aéroports de plus (par exemple), notre programme saurait gérer sans soucis ces ajouts dans notre base de données.

Il est à savoir que **pour gérer les quelques champs vides concernant les coordonnées GPS de certains aéroports, nous avons opté pour une valeur par défaut égale à 0** qui remplit ces champs ne pouvant être renseignés. Nous avons jugé cette valeur explicite pour faire savoir à l'utilisateur que la donnée est inconnue.

Enfin, les données demandant le plus de réflexion quant à leur stockage sont les vols (fichier "flights.csv") à cause de leur nombre conséquent. Nous avons voulu dès le début les stocker dans une **table de hachage** (en hachant par iata code de l'aéroport de départ), afin d'éviter un temps d'accès aux données de vol trop long si nous les avions stockées dans une simple liste chaînée (complexité de recherche linéaire). Nous croyions qu'en hachant les vols selon l'iata code de l'aéroport de départ, nous allions diviser le temps d'accès par 300 environ (sachant que le nombre d'aéroports est de 322). **Cependant, lors des phases de test de l'efficacité de la fonction de hachage, nous avons rapidement remarqué une particularité troublante dans les données de vols : seuls 20 à 50 aéroports parmi les 322 au total figurent comme aéroport d'origine pour les 58492 vols recensés.** En analysant de plus près ces données, nous en avions conclu que seuls les vols au départ des grands aéroports des États-Unis ont été recensés en 2014.

C'est pourquoi nous nous sommes ravisés sur la fonction de hachage et nous avons même décidé de **réaliser 4 tables de hachage distinctes, stockant chacune les vols hachés selon la date (pour la 1ère), l'iata code de la compagnie aérienne (pour la 2ème), l'iata code de l'aéroport de destination (pour la 3ème) et de départ (pour la 4ème).** Nous avons considéré ce choix comme un excellent compromis entre espace mémoire occupé et temps d'accès aux données, **d'autant plus que nous sommes passés par pointeur de structure flight pour stocker les données de vols dans les tables de hachage, si bien que nous payons juste le prix de 1500*espace_mémoire_occupé_par_un_pointeur, soit 1500*8 octets = 12 ko (environ), pour chaque table de hachage supplémentaire, étant donné que nous stockons une seule fois les données de vol en mémoire (pas de duplication de données)**. En effet, toutes les requêtes sauf 4 (celles qui n'ont pas de clé à renseigner) pourront profiter de l'accès rapide offert par les tables de hachage (car donnant une clé primaire dans leur argument). Nous avons passé notre programme sous Valgrind pour voir l'espace mémoire occupé par les listes chaînées de compagnies aériennes, d'aéroports et les 4 tables de hachage de vols, et nous avons vu que **10 Mo étaient occupés en mémoire RAM**, ce qui est

entièrement acceptable au vu des capacités de stockage de mémoire vive des ordinateurs d'aujourd'hui (moyenne de 4 à 8 Go de RAM). **De plus, le choix de structure de données avec 4 tables de hachage permet de simplifier l'écriture de la plupart des requêtes.**

Partie BONUS sur les fonctions de hachage : Nous avons rédigé une procédure de test des fonctions de hachage pour vérifier le nombre de collisions et la bonne répartition des vols dans les tables de hachage. **Cette procédure de test s'appelle "hash_tests" et se trouve à la ligne 104 du fichier "hash_functions.c".** Nous tenons à préciser que si vous souhaitez analyser finement les tables de hachage que nous avons créées (nombre de collisions, nombre de vols contenus à chaque indice de chaque table, etc…), vous pouvez simplement faire appel à cette procédure de test dans le main() et commenter les lignes dédiées aux requêtes pour afficher uniquement ces tests (qui affichent énormément de lignes, attention !).

II - Nos choix de conception algorithmique

La lecture des données est effectuée via le fichier "read_data.c", qui contient les procédures nécessaires à la lecture des fichiers.csv, à leur rangement dans les structures de données adéquates et les procédures d'affichage des données sous forme générique. **Pour lire les fichiers.csv, nous avons utilisé uniquement la fonction standard de "stdio.h" : fscanf() avec les expressions régulières ("regex").**

Les requêtes ont été basées majoritairement sur l'algorithmique des listes chaînées (recherche, parcours, etc…). Cependant, la complexité opératoire de certaines fonctions de requête n'est pas extraordinaire : Certaines requêtes sont améliorables. Aussi, des structures de données temporaires ont été créées pour certaines requêtes afin de faciliter leur écriture.

À savoir que l'ajout, la suppression et l'impression d'une liste chaînée de vols est générique pour chaque table de hachage. Pour cela, il suffit de spécifier un entier défini explicitement via des #define dans "data_structures.h".

Enfin, l'Interface Homme Machine (IHM), codée dans le fichier "menu.c" et "main.c", répond classiquement aux requêtes utilisateur comme sous SQL. Elle a cependant été codée spécifiquement pour interpréter les requêtes propres à ce projet de programmation, donc elle n'est absolument pas générique et adaptable à d'autres bases de données. Ici également, nous n'avons utilisé que la fonction standard de "stdio.h" : scanf(), avec l'expression régulière "%[^\n]s" qui signifie « lecture de tous les caractères renseignés par l'utilisateur jusqu'à l'appui sur la touche "entrée" », pour gérer les requêtes renseignées par l'utilisateur via l'entrée standard.

Pour traiter chaque mot renseigné par l'utilisateur dans sa requête : « commande » « arg1 » « arg2 »…, **nous avons dû coder notre propre fonction de séparation de mot, nommée "isolate_first_word" (ligne 67 de "menu.c"),** qui prend en argument un pointeur de chaîne de caractère pour la faire avancer au mot suivant à l'issue de l'exécution de cette fonction, et qui retourne le premier mot (à gauche) de cette chaîne de caractère. On entend par mot tous les caractères séparés par un espace. **Également, une procédure de décomposition de date "date_decomposition" a été créée (ligne 89 de "menu.c") afin d'interpréter une date au format "MM-DD".**

III - Fonctionnement de notre programme

Un fichier Makefile automatisant la compilation a été rédigée. Dés lors la commande "make" exécutée, le Makefile va créer dynamiquement les dossiers "obj" et "bin", puis il va générer les fichiers objets (.o) à partir des fichiers sources (.c) et les ranger dans le dossier "obj", et il produira enfin l'exécutable à partir de tous les fichiers objets et le rangera dans le dossier "bin". *La commande "make clean" permet de supprimer l'exécutable, les fichiers objets et les dossiers "obj" et "bin".*

Notre programme supporte l'entrée standard : requêtes utilisateurs renseignées manuellement ou par redirection via un fichier.txt. Les requêtes supportées avec leur format sont indiquées clairement dans le README.md, ainsi qu'à l'exécution du programme ou en tapant la commande "help". À savoir que le **temps d'exécution du programme est de 89 millisecondes** avec le fichier "requetes.txt" disponible dans le dossier "data" de notre dépôt GIT.

IV - Limites de notre programme

Le principal manque à notre application d'analyse des 58492 vols ayant eu lieu aux États-Unis en 2014 est la requête "find-multicity-itinerary …". En effet, elle n'est pas codée et si l'utilisateur renseigne cette requête sur le terminal, il lui sera signifié une entrée invalide ("Invalid input ! …"). La complexité de cette requête et le temps pressant nous ont amenés à faire ce choix pour privilégier la qualité du code, la gestion des cas d'erreur et la qualité d'exécution de notre programme.

La vérification de la qualité de notre code avec "clang-tidy" & "cppcheck" a été faite. Néanmoins, quelques warnings de clang-tidy persistent à cause de la non utilisation des versions plus sécurisées des "scanf()" & "fscanf()", qui sont "scanf_s()" & "fscanf_s()". Nous avons pris le parti de laisser les fonctions standards et de ne pas utiliser leurs homologues plus sécurisées, car trop récentes (2019) selon nous pour assurer la compatibilité de notre programme sur les anciennes versions Linux. **La vérification de plagiat avec "moss" n'a cependant pas été faite**, mais à la compilation, il n'y a **aucune erreur ni aucun warning** et tout a été passé au formatage sous "clang-format" avec le fichier ".clang-format" de M. Jérémie DEQUIDT, disponible sur son dépôt GIT exemple. De plus, **aucune fuite mémoire n'est à déclarer sous "valgrind".**

Enfin, nous avons testé la majorité des cas d'erreur au moment où l'utilisateur renseigne ses requêtes, même par la redirection de l'entrée standard avec un fichier.txt. Ainsi, les issues critiques à notre programme sont grandement limitées, **mais nous ne pouvons pas garantir la totale sécurité à l'exécution. De rares cas d'erreur peuvent probablement être critiques !**

CONCLUSION

Ce projet de Programmation Avancée nous aura permis de mettre en application les riches connaissances accumulées lors du Semestre 6. Il constitue une synthèse des nombreux concepts appris en C et en programmation générale :

- **Connaissances basiques en C :** Boucles, tableaux, pointeurs…
- **Structures de données :** Primitives, structures personnalisées à un type particulier (ex : "flight"), listes chaînées, tables de hachage…
- **Gestion des fichiers :** Ouverture, lecture, fermeture.
- **Programmation modulaire :** Fichiers sources (.c), headers (.h), objets (.o) et exécutable (binaire) avec le Makefile qui automatise la compilation (commandes "make" & "make clean").
- **Utilisation d'un gestionnaire de versions : GIT**, avec un dépôt conforme aux véritables projets qui peuvent être menés dans le monde professionnel :
  - Fichiers "Makefile", ".clang-format", ".gitignore", "README.md" & "Rapport.pdf" à la racine du dépôt.
  - Dossier "data" contenant les données utilisées dans le programme & un fichier requetes.txt pour tester les requêtes en batterie.
  - Dossier "src" contenant les fichiers sources (.c).
  - Dossier "includes" contenant les fichiers headers (.h).

- **Qualité du code rédigé :** Respect des conventions de formatage (clang-format), bannissement de la duplication de code et des "magic numbers", noms des variables explicites & commentaires fréquents et pertinents.
- **Qualité de l'exécution :** Aucun "segmentation fault", aucune fuite mémoire avec Valgrind & exécution conforme avec ce qui est attendu + application plutôt permissive avec ce que renseigne l'utilisateur : la plupart des cas d'erreur sont gérés proprement.

Nous avons réalisé un programme qui répond au cahier des charges, disposant de toutes les requêtes demandées (et fonctionnelles), sauf la dernière (complexe), et avec une interface agréable pour l'utilisateur.