Institut de Technologie
Enseignement de Promotion sociale
iTech

mars 2021

Programmation procédurale

Dossier 1

Création d'une base de données de clients

Etudiante : Laura BINACCHI
Professeur : Eric BOSLY
Cours : Programmation procédurale 2020-2021

Table des matières

1 Cahier des charges

Sur base du schéma de base de données présenté ci-dessous et analysé au cours, concevoir et réaliser un programme en C sous CodeBlocks en mode console, géré par des menus simples. La base de données physique sera constituée d'un fichier binaire unique contenant toutes les données et les index nécessaires. Un fichier de trace listera les opérations réalisées sur la base.

Le logiciel devra supporter :

- La création de la base de données
- L'importation des 6 tables fournies sous forme de fichier .csv
- L'exportation de ces 6 tables sous forme de fichier .csv
- La création des index nécessaires aux requêtes précisées
- Des écrans permettant de lister l'information par recherche séquentielle pour les tables de codes
- Des écrans permettant d'afficher par recherche dichotomique
  - Une personne donnée par sa clé primaire
  - Une compagnie donnée par sa clé primaire
- Au moins un écran permettant d'afficher via une liste chainée en mémoire
  - Les personnes travaillant pour une compagnie donnée par sa clé primaire
  - Les compagnies appartenant à un groupe donné
- Au moins un écran utilisant un index binaire sur disque dans la recherche
  - Les personnes travaillant pour une compagnie donnée par sa clé primaire
  - Les personnes dont le nom commence par une chaine donnée
- Les listes chainées pourront être triées ascendant ou descendant
- Deux rapports répondants aux requêtes précisées ci-dessous

Pour réduire la portée du projet, celui-ci sera essentiellement «read-only», une fois les données importées dans la base.

On se concentrera sur :

- la création d'un fichier base de données unique, comme spécifié ultérieurement
- la recherche dichotomique pour les clés primaires de person et company
- la création d'un index sur la clé étrangère id_cpy de la table person
- la création d'un index sur le nom de famille nm_lst de la table person
- la génération de rapports, sous forme de fichiers textes formatés
- l'affichage en mode console des données via des listes chainées

On ne demande pas :

- de modifier les données, mode «read-only»
- de modifier la structure et de rééquilibrer les arbres
- une interactivité poussée, les écrans en mode console seront propres et bien alignés mais simples

On demande un rapport de détail et un rapport agrégé, par exemple :

Rapports de détail :

- Liste de toutes les personnes travaillant pour chaque compagnie d'un groupe
- Liste de toutes les compagnies appartenant à un groupe, par pays

Rapports agrégés :

- Nombre de personnes par groupe (somme des nr_emp) et nombre de personnes connues dans la DB (comptage des records personnes liés au groupe)
- Valeur totale des actions détenues par les employés pour un groupe donné, éclaté par par genre et niveau de fonction.

2 Description du programme

2.1 Fichier de base de données

Le programme est centré sur l'exploitation d'un fichier de base de données. Cette base de données est décrite dans le fichier *catalog.h*. Le catalogue définit les différentes structures composant la base de données : les tables (*country*, *job*, *industry*, *group*, *company* et *person*), les index numériques et les index alphanumériques. Il définit également les tailles des champs de ces enregistrements [1].

Le catalogue définit le nombre d'emplacements réservés pour chaque table. Ils serviront à la création des enregistrements "vides" [2] en base de données, à la fois les enregistrements des tables et des index (un index portant sur la table *person* doit avoir le même nombre d'emplacements réservés que cette table). Il définit où enregistrer les fichiers créés et où trouver les fichiers d'import.

Le catalogue définit enfin les structures de métadonnées sur les tables et les index dont les valeurs sont assignées dans le fichier *catalog.c*. Ces métadonnées regroupe les informations spécifiques à chaque table comme le nom ou le préfixe à enregistrer en base de données et permettent la généricité de certaines fonctions. Par exemple, chaque table a une fonction d'import qui est appelée dans un boucle plutôt que de dupliquer le code d'import. Cette généricité rendra aussi plus facile l'ajout d'une nouvelle table.

2.2 Informations sur la base de données

Certaines informations sur le fichier de base de données sont chargées en mémoire RAM. Elles sont regroupées dans la structure *db* définie dans le fichier *database.h*. Cette structure permet de passer un seul paramètre, son pointeur, aux différentes fonctions du programme pour accéder :

- au header de la base de données : ce header est enregistré au début du fichier de base de donnée et reprend les informations essentielles à l'exploitation du fichier (taille, nombre d'emplacements réservés et d'enregistrements dans chaque table, offsets des tables et des index) ;
- aux pointeurs de fichiers : fichier de base de données, de logs et fichier csv si un fichier d'import est ouvert ;
- aux tables de codes chargées en mémoire : elles permettent un accès direct aux enregistrements (par identifiant) ou à toute la table plus rapidement que par une lecture dans le fichier de base de données.

Le header et les tables de codes sont chargée en mémoire par la fonction d'ouverture de fichier implémentée dans *open_ close.c* à chaque fois que le fichier est ouvert en mode *read* ou *append*. En mode *write*, ce sont les informations du header en mémoire RAM qui sont écrites dans le fichier de données.

2.3 Menus

Conformément au cahier des charges, j'ai implémenté un menu en console afin d'accéder au fichier de base de données. Ce menu est programmé de manière déclarative : toutes les entrées de menu sont décrites dans

1. Ces tailles ont été déterminées en lançant l'exécutable `csv_field_length.sh`. Pour chaque fichier d'import, l'exécutable affiche la plus longue ligne et sa taille (utilisée pour définir la taille du buffer de lecture des fichiers *csv*) et le plus long de chacun des champs et sa taille. Ces tailles sont adaptées avec ajout d'un filler pour faciliter la lecture du fichier binaire dans un éditeur hexadécimal.
2. Un enregistrement n'est jamais totalement vide puisqu'il a au minimum un type (ou préfixe).

un tableau et sont lancées par une fonction unique. Cette approche permet d'ajouter facilement une nouvelle entrée sans devoir modifier le code qui affiche les menus, attend une entrée de l'utilisateur et appelle les fonctions correspondantes.

Les menus sont divisés en deux catégories : ceux du mode utilisateur et ceux du mode d'administration. Le mode utilisateur regroupe les fonctionnalités d'exploitation de la base de données qui ne nécessitent qu'un accès *read-only* au fichier. Le mode d'administration regroupe les fonctionnalités nécessitant un accès en écriture au fichier. Le mode d'accès au programme est défini par un argument passé au lancement de l'exécutable [3].

Le fichier *menu.h* définit la structure d'une entrée de menu avec le nom du menu et le pointeur de la fonction à appeler correspondant à ce menu. Un menu est composé d'une liste de ces entrée, du nombre d'entrées composant le menu et d'un titre. Les différents menus sont rassemblés dans un tableau. Le fichier *menu.c* assigne des valeurs à ce tableau et implémente la fonction qui affiche le menu et appelle la fonction associée à l'entrée choisie par l'utilisateur.

2.4 Administration du fichier de base de données

Le fichier *admin.c* implémente les fonctions de création et de suppression du fichier de base de données. La création du fichier crée un nouveau fichier "vide" et écrase tout fichier existant si il y en avait un. Tout l'espace nécessaire au remplissage des données est réservé et rempli de `0` à l'exception du type d'enregistrement. Le header est rempli avec le nom de la base de données, le nombre d'emplacements réservés pour chaque table et les offsets des tables et des index, en mémoire et dans le fichier de données. Lors de la suppression du fichier via l'interface d'administration, le header gardé en mémoire est remis à zéro, ce qui ne serait pas le cas si le fichier était supprimé manuellement alors que le programme est en cours.

Ce fichier implémente aussi les fonctions d'import et d'export de données. Pour chaque table, l'import ouvre le fichier *csv* d'import et écrit les données lues dans le fichier de données à partir de l'offset approprié. Le nom de ces fichiers et leur emplacement est défini par le catalogue dans les métadonnées des tables. L'import lance automatiquement la génération des index car ils sont nécessaires au fonctionnement de certaines fonctions d'exploitation de la base de données. Le fonctionnement des index est décrit au point suivant.

La fonction d'export recrée les fichiers *csv* à partir du fichier de base de données. Il est essentiellement utilisé pour vérifier la validité des données importées (les fichiers d'import et d'export doivent être identiques).

Le fichier *admin.c* implémente enfin une fonction d'affichage des métadonnées de la base de données, i.e. des informations contenues dans le header. Cette entrée de menu est la seule qui soit partagée entre le mode d'administration et le mode d'exploitation de la base de données.

2.5 Index numériques, alphanumériques et arbre binaire

Tous les index sont créés de la même manière que les tables à partir d'un tableau de métadonnées défini dans le catalogue. La création des index numériques est dissociée de celle des index alphanumériques car

3. Si le premier argument passé au main est la chaîne de caractères "admin", le programme est lancé en mode administrateur. Dans tous les autres cas, le programme est lancé en mode utilisateur.

leurs structures sont différentes et l'index alphanumérique nécessite la création d'un arbre binaire. Elles sont respectivement implémentées dans les fichiers *num_ index.c* et *alpha_ index.c*.

Le mécanisme de création de l'index est commun aux deux types. Les fonctions de création d'index constituent ceux-ci à partir de la valeur sur laquelle porte l'index (identifiant de groupe ou de compagnie pour les index numériques et nom de famille des personnes pour l'index alphanumérique). Cette valeur est lue par la fonction de lecture définie dans les métadonnées : e.g. la lecture de *id_ group* pour la constitutions de l'index *company by group id*.

Après avoir constitué la liste de valeurs de l'index, cette liste va être triée par une fonction de tri *quick sort* détaillée au point suivant. Cette liste triée est écrite dans le fichier de données.

Les index numériques crées de cette façon sont directement exploitables par une fonction de recherche dichotomique. Les index alphanumériques doivent quant à eux être représentés sous forme d'un arbre binaire équilibré dont la fonction de création est implémentée dans le fichier *alpha_ index.c* selon l'algorithme récursif vu au cours. Chaque tuple de l'index enregistre l'offset des nœuds gauche et droit. L'offset du tuple racine de l'arbre est écrit dans le header du fichier de données et gardé en mémoire RAM. Il constitue le point d'entrée des recherches utilisant l'index alphanumérique.

2.6 Tri *quick sort*

La fonction de tri *quick sort* vue au cours est implémentée dans le fichier *sort.c*. La signature de la fonction est la même que celle de la fonction *qsort* de la librairie standard *stdlib* afin de comparer facilement le temps d'exécution de la fonction implémentée par le programme et celle de la librairie standard.

La particularité de cette fonction est de trier un tableau d'éléments dont le type n'est pas connu. Le pointeur du tableau est passé en paramètre à la fonction avec le nombre d'éléments qui le composent et la taille d'un élément. Cette particularité rend impossible l'accès direct à un élément du tableau par un appel classique de type `tableau[i]`. Cet accès est implémenté par la fonction privée `void *_elem(void *ptr, size_t element_size, size_t i)`.

Les fonctions utilitaires privées sont appelées par l'intermédiaire de macros pour pouvoir être appelées sans leur passer des arguments qui sont toujours les mêmes. Par exemple, la fonction qui retourne un élément du tableau à un index donnée est définie comme suit :

```
#define QSORT_ELEM(__i)         _elem(ptr, element_size, __i)
```

Elle sera appelée uniquement dans la fonction privée `_quick_sort` ou `ptr` (le pointeur du tableau à trier) et `element_size` sont connus et ne changent pas. L'index `__i` qui lui est différent à chaque appel est passé en paramètre à la macro.

Enfin, les éléments du tableau sont comparés par la fonction de comparaison passée en paramètre au *quick sort*. Si besoin, ils sont intervertis par la fonction `_swap`.

2.7 Fonctions de recherches

Les fonctions de recherches sont des fonctions d'exploitation de la base de données. Elles sont donc accessibles uniquement via le mode utilisateurs.

Les fonctions de recherches à proprement parler (séquentielle, dichotomique et par index) sont séparées des fonctions qui les exploitent, que ce soit pour l'affichage en console ou la génération de rapport. Ces fonctions de recherches sont implémentées dans le dossier *search*, les affichages en console dans le dossier *display_search* et les rapports dans le dossier *report*.

Les fonctions de recherches génériques du dossier *search* revoient des listes chainées de structures de type `search_result` définie dans le fichier *search_result.h*. L'implémentation des listes chaînées est détaillée plus loin dans ce travail.

Il est enfin possible de trier les listes de résultats grâce à la fonction de tri implémentée dans le fichier *search_result.c*. Cette fonction générique prend en paramètres une liste de résultats et une fonction de comparaison. Elle utilise l'algorithme *quick sort* après transformation de la liste chaînée en tableau.

2.8 Recherche par accès direct

Les recherches par accès direct ne sont pas utilisées directement dans les menus de recherches mais par l'intermédiaire d'autres recherches. Par exemple, la génération du rapport des compagnies par groupe fait une recherche directe de groupe par l'identifiant cherché par l'utilisateur. La recherche directe est alors une recherche dans le buffer chargé en mémoire RAM.

L'implémentation d'une fonction de recherche directe ne nécessite pas d'implémentation particulière. En effet, il suffit d'accéder directement au tuple cherché par un appel classique dans le buffer de type `buffer[id-1]`. Pour une recherche directe dans le fichier de données, il aurait fallu implémenter une lecture dans le fichier de données sur base d'un offset calculé à partir de l'id recherché, de l'offset de la table et de la taille du tuple.

2.9 Recherche séquentielle

Une fonction générique de recherche séquentielle est implémentée dans le fichier *sequential_search.c*. Elle constitue la liste de résultats à partir de la fonction de comparaison définies dans les métadonnées des tables.

Elle est exploitée par les recherches implémentées dans *display_sequential_search.c* : recherches de pays, de job, d'industrie et de groupe par début de chaîne de caractère. Ces recherches demandent à l'utilisateur la chaîne de caractère cherchée pour rechercher :

- un pays par début de nom, de zone ou de code ISO (implémentation dans *country.c*)
- un job par début de nom, de niveau ou de département (implémentation dans *job.c*)
- une industrie par début de nom ou de secteur (implémentation dans *industry.c*)
- un groupe par début de nom (implémentation dans *group.c*)

Toutes ces recherches sont insensibles à la casse.

La liste de résultats trouvée est affichée par une fonction générique de pagination implémentée dans *ui_utils.c*. Elle est triée par identifiant puisque les tuples sont parcourus séquentiellement et que les base de données sont triées sur base des identifiants. Elle est affichée à l'endroit ou à l'envers selon le choix de l'utilisateur (à l'endroit par défaut).

2.10 Recherche dichotomique

Le fichier *binary_search.c* implémente la fonction générique de recherche dichotomique selon l'algorithme récursif vu au cours. Cet algorithme est utilisé pour les recherches par identifiant. Puisqu'il est unique, un seul résultat est retourné : l'offset du tuple trouvé ou celui du tuple le plus proche.

La recherche dichotomique est utilisée par la fonction générique `display_search_by_id` du fichier *display_binary_search.c*. Un identifiant est demandé à l'utilisateur et passé en paramètre à la recherche dichotomique. Si l'utilisateur entre `0`, le programme revient au menu principal n'effectue pas la recherche et l'utilisateur peut revenir au menu principal. Sinon, il affiche le tuple trouvé ou les identifiants des trois tuples les plus proches de celui cherché.

Il est à noter que la recherche de groupe par identifiant aurait plutôt pu être implémentée par un accès direct dans le buffer de groupes.

2.11 Recherche via index numérique

Les index numériques sont exploités par la fonction de recherche implémentée dans *num_index_search.c*. Cette fonction trouve la première occurrence correspondant à la recherche dans l'index et retourne la liste de résultats à partir de celle-ci.

Les recherches par index numérique affichées en console sont implémentées par une fonction générique dans le fichier *display_num_index_search.c* qui permet de chercher des personnes par identifiant de compagnie. L'identifiant est demandé à l'utilisateur est la liste est affichée par la fonction de pagination. Elle est triée par ordre alphabétique sur le nom de famille des personnes et est affichée à l'endroit ou à l'envers selon le choix de l'utilisateur.

Les rapports utilisent également la recherche numérique :

- Le rapport de compagnies par identifiant de groupe companies utilise l'index *company by group id*
- Le rapport des employés d'un groupe classés par compagnie utilise l'index *company by group id* pour trouver la liste des compagnies du groupe et l'index *person by company id* pour trouver la liste des employés de chaque compagnie
- Le rapport du compte des employés d'un groupe utilise les mêmes index pour trouver les employés du groupe
- Le rapport sur les valeurs des actions détenues par les employés d'un groupe utilise les mêmes index pour trouver les employés du groupe

Dans tous les cas, les listes de personnes sont triées par ordre alphabétique par nom de famille. Le programme n'intègre toutefois pas le tri dans la recherche par index numérique car une autre utilisation pourrait ne pas vouloir trier la liste ou la trier selon un autre champ.

2.12 Recherche via index alphanumérique

La recherche par index alphanumérique est implémentée dans le fichier *alpha_index_search.c*. De la même manière que la recherche par index numérique, elle retourne une liste de résultats à partir de la première occurrence correspondant à la recherche trouvée dans l'arbre binaire.

Cette recherche est utilisée dans *display_alpha_index_search.c* pour afficher la liste de personnes par début de nom. La liste n'est pas triée puisque l'index alphanumérique trie déjà les personnes par nom de famille. Ce tri est donc conservé à l'affichage et peut être inversé selon le choix de l'utilisateur.

2.13 Liste doublement chaînée

Le programme implémente une structure générique de liste doublement chaînée dont la structure est définie dans *linked_list.h*. Cette liste est composée d'une donnée : un pointeur de *void* qui peut donc correspondre à n'importe quel type de donnée ou de structure. Elle contient le pointeur de l'élément suivant et de l'élément précédent.

Le fichier *linked_list.c* implémente les fonctions d'ajout d'un nouvel élément à la liste et de suppression de la liste, avec ou non une libération de la mémoire des données contenues dans la liste.

2.14 Logger

Toutes les fonctions d'exploitation de la base de données écrivent des informations dans un fichier de log. Ce logger est implémenté dans le fichier *logger.c* : il permet d'afficher un message d'erreur à la façon de `perror` en exploitant la valeur de `errno` ou d'afficher toute autre information précédée de l'endroit d'où provient le log.

3 Structure de fichiers

En résumé de la description du programme, voici la structure de fichiers de mon programme. Le dossier *src* n'est pas développé pour ne pas répéter la structure de fichiers déjà détaillée dans le dossier *include*. Seuls certains fichiers sont uniquement des headers et ne contiennent pas de prototype de fonction à implémenter dans les sources.

```
- data_clients (dossier créé par le programme)
    db_clients.dat (fichier de données binaires)
    db_clients.log (fichier de logs généré par l'application)
- data_export (dossier créé par le programme lors de l'export de données)
    Company_Export_Datetime.csv (date et heure auxquelles le fichier a été généré)
    Country_Export_Datetime.csv
    Group_Export_Datetime.csv
    Industry_Export_Datetime.csv
    Job_Export_Datetime.csv
    Person_Export_Datetime.csv
- data_import (dossier qui doit être présent avec les six fichiers d'import
               au moment de l'import dans le fichier binaire de données)
    DB_Company.csv
    DB_Country.csv
    DB_Group.csv
    DB_Industry.csv
    DB_Job.csv
    DB_Person.csv
- include (headers des fichiers sources)
    - db_file
        admin.h (fonctions d'administration de la base de données)
        alpha_index.h (création d'index alphanumérique utilisant un arbre binaire)
        catalog.h (définitions des structures des tuples et des métadonnées)
        database.h (définition des structures de données gardées en mémoire RAM)
        header.h (création du header pour une base de données vide)
        num_index.h (création d'un index numérique)
        open_close.h (fonctions d'ouverture et de fermeture du fichier de données)
    - display_search (recherches dont les résultats sont listés en console)
        display_alpha_index_search.h
            recherche de personnes par nom de famille (tri par nom de famille)
        display_binary_search.h
            recherche d'un groupe par id
            recherche d'une compagnie par id
            recherche d'une personne par id
        display_num_index_search.h
            recherche de personnes par id de compagnie
        display_search.h (rassemble les headers pour simplifier les inclusions)
        display_sequential_search.h
            recherche par début de chaîne de caractère dans les tables de codes
            (pays, job, industrie, groupe)
```

```
    - report
        companies_by_group.h
            compagnies d'un groupe (par id) groupées par pays (rapport de détail)
        people_by_group.h
            employés d'un groupe (par id) groupés par compagnie (rapport de détail)
        people_count_by_group.h
            pourcentage d'employés connus d'un groupe par id (rapport agrégé)
        people_shares_sum.h
            somme d'actions détenues par les employés d'un groupe présentant
            les statistiques par niveau de fonction et par genre (rapport agrégé)
        report_file.h (création d'un fichier texte de rapport)
        report.h (rassemble les headers pour simplifier les inclusions)
    - search
        alpha_index_search.h (recherche par index alphanumérique générique)
        binary_search.h (recherche dichotomique générique)
        num_index_search.h (recherche par index numérique générique)
        search_result.h (structure des résultats de recherche et fonction de tri)
        sequential_search.h (fonction générique de recherche séquentielle)
    - table (fonctions de manipulation des tables)
        company.h
        country.h
        group.h
        industry.h
        job.h
        person.h
    - ui
        menus.h (menus déclaratifs)
        ui_utils.h (interface utilisateur)
            récupération d'inputs : nombre, chaîne de caractères, oui/non
            affichages : liste paginée, détails d'un record (ou résultat
                le plus proche), pause du programme
    - utils
        linked_list.h (gestion de liste doublement chaînée générique)
        logger.h (log d'information dans le fichier db_clients.log)
        preprocess_string.h (macro de transformation d'un nombre en string)
        sort.h (algorithme de tri quick sort)
        string_utils.h (fonctions utilitaires sur les chaînes de caractères :
            comparaisons, remplacement d'un caractère, mise en minuscules)
        system.h (fonction dépendant du système d'exploitation)
+ out (dossier dans lequel sont placés les fichiers objets compilés)
+ src (implémentations des headers, contient le main)
+ tests (fichiers de tests)
Makefile (fichier utilisé pour compiler le projet)
```

4 Structure du fichier database

Le fichier de données *db_ clients.dat* est un fichier binaire d'une taille totale de 208 MB (208 224 128 bytes) composée de :

- un header (128 bytes)
- 100 tuples dans la table *country* (100×64bytes $= 6400$bytes)
- 200 tuples dans la table *job* (200×96bytes $= 19200$bytes)
- 100 tuples dans la table *industry* (100×64bytes $= 6400$bytes)
- 3000 tuples dans la table *group* (3000×64bytes $= 192000$bytes)
- 100 000 tuples dans la table *company* (100000×288bytes $= 28800000$bytes)
- 500 000 tuples dans la table *person* (500000×224bytes $= 112000000$bytes)
- 100 000 tuples dans l'index numérique *company by group id* (100000×32bytes $= 3200000$bytes)
- 500 000 tuples dans l'index numérique *person by company id* (500000×32bytes $= 16000000$bytes)
- 500 000 tuples dans l'index alphanumérique *person by lastname* (500000×96bytes $= 48000000$bytes)

5 Tests

Tous les tests du dossier *tests* peuvent être compilés et lancés par le *Makefile* avec la commande `make tests`. Les tests d'intégration peuvent être lancés avec la commande `make run_integration_tests` et les tests unitaires avec la commande `make run_unit_tests`.

5.1 Import des données

Le processus complet de la création du fichier de données est testé par le fichier *test_ create_ db.c* (test d'intégration) : création du fichier de données vide, import des données à partir des *csv*, export, affichage des métadonnées et log des informations dans *db_ client.log*.

Comparer la taille réelle du fichier avec la taille calculée par le programme m'a permis de trouver une erreur au niveau de la création des index vides. Après correction, le fichier de données fait bien la taille calculée par le programme. Les offsets des différentes tables correspondent bien aux offset théoriques.

L'intégrité des données est vérifiée par comparaison avec les fichiers exportés. Les fichiers exportés sont légèrement plus petits que les fichiers d'import. En effet, les fichiers d'import contiennent un retour charriot qui a dû être supprimé pour le bon fonctionnement des affichages sous une distribution Linux.

La commande `diff -strip-trailing-cr import data` (où `import` et `export` sont remplacés par les noms des fichiers à tester) permet de vérifier que la présence ou non des retours charriots est bien la seule différence. Le fichier d'import des compagnies présente une différence supplémentaire : la présence d'espaces dans le champ *am_ val*. L'ajout de l'argument `-w` à la commande `diff` permet de confirmer que c'est bien la seule différence.

5.2 Interface utilisateur

Tous les éléments d'interface en console ont été testés en lançant le programme et en vérifiant que tout se déroulait correctement : menus, affichage des résultats, des listes, etc.

La navigation dans le menu a été testée en vérifiant évidemment que le menu affiché correspondait bien au menu choisi par l'utilisateur mais aussi en entrant des entrées non valides. Le menu tout comme les entrées de menu qui attendent un entier positif ne prennent en compte que le premier entier valide entré [4]. Le buffer de l'entrée standard est ensuite vidé pour ne pas tenir compte de tous les caractères suivants.

Le programme pourrait être amélioré en répétant cette opération avant la lecture d'une entrée utilisateur : dans l'état actuel du programme, il est possible d'entrer une donnée à un moment où elle n'est pas attendue et elle sera prise en compte par la prochaine fonction attendant une entrée. Si cette entrée n'est pas valide, elle sera ignorée avec affichage du message d'erreur. Sinon, elle est prise en compte et le programme effectue une action qui n'a pas forcément été demandée.

4. Cette fonctionnalité est implémentée par la méthode `get_uns_input` du fichier *ui_ utils.c*

5.3 Cas d'erreur

Les cas d'erreurs ont été testés en provoquant volontairement les erreurs, e.g. en lançant l'import de données sans fichier d'imports présents. Les possibilités de ce type d'erreurs ont été limitées au maximum en anticipant la possibilité qu'elle advienne, e.g. si le répertoire *data_ export* n'existe pas, il est créé avant la création de fichiers d'export. S'il est impossible de prévenir ces erreurs, elles sont gérées par l'affichage d'un message d'erreur et le log de l'erreur.

Le fichiers de logs est un bon outil pour vérifier le bon fonctionnement du programme. L'écriture des logs appropriés a été testé tout au long de l'écriture du programme, si possible en provoquant volontairement les erreurs, sinon en modifiant les fonctions.

5.4 Tests unitaires

5.5 Fichier de logs

5.6

6 Problèmes rencontrés

6.1 Compatibilité entre Windows et Linux

Durant le développement de l'application, j'ai été confrontée à de nombreux bugs dont certain ont été plus difficiles à trouver que d'autres. Dans la plupart des cas, les messages d'erreurs étaient assez clairs. Un des problèmes les plus difficiles à debugger auxquels j'ai été confrontée était un problème d'affichage : lors de l'affichage des listes de résultats, certains caractères et parfois certaines lignes entières étaient effacées alors que les données semblaient correctes.

Le problème était en fait un problème de compatibilité entre Windows et Linux : le retour à la ligne se fait sous Windows avec un retour charriot supplémentaire. J'ai réglé se problème dans les fonctions d'import des tables en testant la présence d'un `\r` à la fin du dernier champ et en le remplaçant par un `0` si nécessaire.

J'ai essayé de rendre mon programme le plus compatible possible avec une distribution Windows. Les fonctions propres au système d'exploitations ont été évitées au maximum. Quand elles étaient nécessaires, elles ont été implémentées dans un `ifdef` qui teste l'OS. Celui-ci est également testé au lancement du programme pour empêcher le programme d'être lancé sous une distribution pour laquelle le programme n'a pas été conçu et pour laquelle il n'a donc pas été testé. Malgré cela, le programme a été testé uniquement sous une distribution Linux et pourrait donc présenter des bugs sous Windows.

6.2 Inclusions circulaires

Certaines difficultés rencontrées sont liées à la taille du projet demandé. C'est la cas des inclusions circulaires : un premier fichier inclut un second qui lui-même inclut le premier. Cette difficulté a pu être contournée la plupart du temps par une modification de l'architecture du programme.

Le seul endroit où j'ai résolu ce problème différemment est dans les fichiers *catalog.h* et *database.h* qui s'incluent mutuellement. J'ai supprimé l'inclusion de *database.h* dans *catalog.h* en faisant une déclaration avancée de la structure `db`, à la manière de la déclaration d'un prototype de fonction avant son implémentation.

6.3 Gestion des erreurs

La gestion des erreurs peut très vite alourdir le code. Elle est pourtant nécessaire à son bon fonctionnement. Par contre, il est impossible de gérer toutes les erreurs et ce n'est d'ailleurs pas souhaitable : certaines erreurs doivent arrêter le programme.

L'endroit du programme où sont gérées les erreurs a été difficile à déterminer. J'ai essayé de le faire de manière cohérente : les affichages des erreurs et les écritures de logs ne sont pas gérées dans les fonctions utilitaires mais dans les fonctions qui les appellent. En effet, les fonctions d'appel peuvent vouloir gérer les mêmes erreurs différemment. J'aurais également pu gérer le log des erreurs dans chaque fonction et l'affichage à l'utilisateur dans les fonctions d'interface utilisateur.

6.4 Généricité du code

Enfin le bon degré de généricité à implémenter n'a pas toujours été facile à déterminer. Une analyse préalable du code m'a permis de déterminer de manière assez juste quand génériciser les fonctions mais je me suis trompée notamment dans la manière de découper les fonctions de recherche. J'avais inclus les affichages en console dans les fonctions de recherches alors que ces fonctions sont aussi appelées pour la génération des rapports, sans affichage des résultats donc.

Implémenter du code générique m'a pris plus de temps que de copier-coller le code des fonctions en l'adaptant. Le faire m'a pourtant permis d'aller beaucoup plus loin dans l'apprentissage du C, avec par exemple l'utilisation de pointeurs de fonctions ou encore avec l'utilisation du *Makefile*, nécessaire pour compiler un projet avec autant de fichiers.