TP Librairie de Chansons

notions : structures, listes chaînées, fichiers

Tous les exercices de ce TP construisent pas à pas un programme complet. Il est indispensable de créer et utiliser un `Makefile`.

On veut réaliser une application de gestion de liste de chansons.
Les chansons seront chargées depuis un fichier texte dans lequel chaque chanson est définie par 4 éléments :
— l'identifiant de la chanson, unique pour chaque chanson : un entier (non signé) ;
— son titre : une chaîne de caractères de longueur quelconque ;
— son (ses) auteur(s) : *idem* ;
— le format du fichier de la chanson (`"WAV"` ou `"MP3"` par exemple) : une chaîne de 3 caractères.

Exercice 1 Prérequis

Structurez l'arborescence ainsi (on suppose que le répertoire principal s'appelle `playlist`) :
— `playlists/Makefile` : le Makefile du projet
— `playlists/include` : contiendra les fichiers d'en-tête `.h` utilisés par le programme de l'application et par les programmes de test ;
— `playlists/src` : contiendra les fichiers de gestion des chansons et listes de lecture ;
— `playlists/tests` : contiendra les fichier source `.c` pour les tests de nos fonctions dans `playlists/src`.
— `playlists/playlist.c` : contiendra notre exécutable final ;

Exercice 2 Structure de donnée pour représenter une chanson

1. Proposez une structure de donnée pour représenter une chanson (unique). Une chanson sera manipulée par un type pointeur sur structure, nommé `song_t`.
   Implantez votre structure de donnée dans un module **song** (`include/song.h` / `src/song.c`) ;
2. Implantez la fonction `song_t song_new( unsigned int id, char *title, char *artist, char *format );`
3. Implantez la fonction `void song_print( song_t s );`
4. Implantez la fonction `void song_delete( song_t s );`
5. Dans votre premier test (`tests/song_test.c`), écrivez un programme principal qui construit une variable chanson, qui l'affiche à l'écran et qui libère la mémoire utilisée.

Exercice 3 Lecture d'une chanson

1. Ajoutez au module **song** la fonction `song_t song_parse( char *line );` qui prend en paramètre une chaîne de caractères encodant toutes les informations d'une chanson, extrait les informations de la chaîne et construit et retourne une valeur de type `song_t` (*cf.* question Quelle valeur que retournera la fonction en cas d'erreur ? Votre fonction `song_parse` pourra en plus afficher la raison de l'éventuelle erreur de lecture rencontrée.

   On supposera que les informations sont encodées dans la chaîne au moyen du format suivant :

   `<rang>\t<format>\t<titre chanson>\t<auteur chanson>`

   où `\t` est le caractère tabulation.

   Exemples de chaînes valides (entre chaque champ, c'est une tabulation `\t`) :

   ```
   4420  AIF   Ready to Start      Arcade Fire
   4     MP3   changer le monde    Jeunes UMP
   ```

   Pour cette question, vous pouvez utiliser les fonctions :
   - `char *strchr(char *str, char c)`, pour renvoyer la prochaine occurrence du caractère `c` dans la chaîne `str` (ou `NULL` si `c` ne figure pas dans `str`) ;
   - `char *strncpy(char * dst, const char * src, size_t n)` pour copier n caractères de la chaine `src` dans la chaîne `dst` (attention : cette fonction n'ajoute pas le caractère de terminaison `\t` à la fin de la chaîne destinataire `dst`).

   Voir les pages de manuel (eg `man strchr`).

   Il est également possible d'utiliser les fonctions suivantes :
   - `char *strtok(char *str, const char *delim)`, pour parcourir la ligne lue mot par mot (difficile) ou
   - `int sscanf(const char *str, const char *format, ...)`, pour convertir une chaîne en nombre entier ;
   - `char *strdup(const char *s)` pour dupliquer une chaîne.

2. Dans votre second test (`tests/song_test_parse.c`), écrivez un programme principal qui vérifie que votre fonction `song_parse` lit correctement une ligne bien formée. La chaine de caractère correcte à parser pourra être initialisée par exemple ainsi :

   `char * chaine_to_parse = "187\tWAV\tyestreday\tThe Beatles";.`

   Recensez aussi les cas d'erreur (chaîne incorrecte) et testez que chacun de ces cas est bien détecté en vérifiant que l'appel de `song_parse` renvoie une erreur (`NULL` avec notre convention). Voici de quoi créer une chaine avec un exemple d'erreur :

   `char * chaine_to_parse = "187\tWAV\tyestredayThe Beatles";` (il manque une tabulation).

   Enfin, complétez votre programme en permettant de parser une chaîne entrée par l'utilisateur au clavier.

Exercice 4 Fichier de chansons

On dispose d'un fichier texte de chansons (une librairie) *music.csv*, dont voici un extrait :

```
3212    MP3   Take On Me    a ha
6365    WAV   Mysterious Ways   Angelique Kidjo
```

Chaque ligne valide contient donc le rang de la chanson, son format audio, son titre et son auteur, séparés par une tabulation (le monde est bien fait !). Attention : ce fichier peut contenir quelques lignes incorrectes (chaîne invalide).

Nous souhaitons compter le nombre de chansons valides et invalides du fichier chanson.

1. Dans un nouveau module **songfile** (fichiers `include/songfile.h` et `src/songfile.c`), écrivez la fonction `int song_file_count( char *filename, int *pNvalid, int *pNinvalid )`.

   Cette fonction *parse* le contenu du fichier nommé `filename`. Elle utilise les deux paramètres pointeurs pour "retourner" le nombre de chansons valides et invalide du fichier. Elle retourne 0 si le fichier a pu être ouvert, un nombre non nul sinon.

   Dans un premier temps, pour vérifier, elle ré-affichera dans le Terminal chaque chanson lue. La ligne qui fait cet affichage pourra plus tard être mise en commentaire.
2. Dans un nouveau test (`tests/test_songfile_count.c`), écrivez un programme principal qui compte et affiche le nombre de chansons valides et invalides trouvées dans un fichier de chansons.

Exercice 5 Liste chaînée générique pour stocker des chansons

On veut pouvoir stocker en mémoire les chansons contenues dans une librairie. Le nombre de chanson est inconnu à l'avance et il convient de pouvoir en ajouter à la volée.

On utilisera donc pour cela une liste chaînée.

De plus, comme nous aurions besoin plus tard de manipuler d'autres listes qui contiennent non pas des chansons, mais d'autres types d'éléments (... des listes "d'artistes" par exemple ...), nous souhaitons implanter les listes chaînées "génériques", capables de stocker n'importe quel type d'éléments. Le module liste générique sera implantée dans les fichiers (`include/list.h` et `src/list.c`).

1. Quel type "générique" choisir en C pour les éléments de notre liste générique ?
2. Dans le module liste, définissez le type "liste chaînée générique" ;
3. Dotez ce module des principales fonctions implantant le type abstrait "liste" : `list_t list_new()`, `int list_empty(list_t)`, `list_t list_add_first(list_t l, void * element)`, `int list_count (list_t)`, `list_t list_delete_first(list_t l)`, `list_t list_delete(list_t)` ;
4. Ajoutez une fonction au module list une fonction `void list_add_last(list_t l, void* element )` qui ajoute un élément à la fin de la liste. Quelle est la complexité de cette fonction ?
5. Ecrivez un nouveau programme test (`tests/test_list_song.c`) qui :
   — crée une liste chaînée vide.
   — ajoute deux chansons quelconques à la liste
   — affiche le nombre d'éléments de la liste
   — supprime la chanson en tête de liste
   — affiche le nombre d'éléments de la liste
   — demande à l'utilisateur de saisir une chanson et l'ajoute **à la fin** de liste
   — affiche le nombre d'éléments de la liste
   — détruit la liste
   — affiche le nombre d'éléments de la liste

Si des erreurs de segmentation surviennent, utilisez l'utilitaire Valgrind ou le débugger gdb pour les comprendre et les corriger.

Exercice 6 Affichage du contenu d'une liste générique. Pointeur de fonction.

On veut doter le module list d'une fonction `list_print` qui affiche les éléments contenus dans une liste.

Comment, dans cette fonction du module liste, afficher un élément alors que la liste générique n'a aucune idée du type effectif de l'élément, qui se cache au bout du pointeur de type `void *` ?

Il est nécessaire que la fonction `list_print` prennent ait connaissance du moyen permettant d'afficher cet élément, adapté au cas par cas au type effectif de l'élément.

En langage C, Le mécanisme qui s'impose est d'ajouter un paramètre de type pointeur-de-fonction.

1. Dans un fichier d'en-tête `callback.h`, définissez le type pointeur-de-fonction nommé `callback_t`. Une fonction de type `callback_t` prendra en paramètre un élément (type `void *`) et retournera un entier.
2. Dans le module song, définissez la fonction-callback `int song_print_callback( void *ptr )`. Cette fonction suppose que `ptr` pointe en fait une chanson, puis se contente d'appeler la fonction `song_print`. Elle retourne 0.

   Notez au passage que le prototype de cette fonction est bien conforme au type type `callback_t`.
3. Dans le module list, ajoutez une fonction `void list_print(list_t l, callback_t print_cb)`, qui applique la fonction `print_cb` à chacun des éléments de la liste, en la parcourant.
4. Modifiez le programme de test (`tests/test_list_song.c`) pour afficher à chaque étape le contenu de la liste.

Comment procéderiez vous pour pouvoir maintenant choisir dans le programme principal d'afficher par exemple uniquement les noms des chansons contenus dans une liste, ou uniquement leur numéro ?

Exercice 7 *Pattern* visiteur

En fait, il y a de nombreuses actions autres que l'affichage qu'on peut avoir besoin d'appliquer à tous les éléments d'une liste générique.

Si la liste contient des chansons, on pourrait par exemple vouloir compter les chansons dont le nom commence par "a" ; si elle contient des "immeubles", on pourrait par exemple vouloir récupérer le nombre total de logements ; etc.

Dans toutes ces situations, c'est toujours le même mécanisme qui est en jeu : il s'agit de parcourir la liste et, pour chacun de ses éléments, d'appliquer une fonction particulière.

Nous allons mettre en place le pattern "visiteur" sur nos listes génériques :

1. Dans le module list, ajoutez une fonction `int list_apply(list_t l, callback_t action_cb.` Cette fonction :
   — parcourt la liste
   — exécute la fonction `action_cb` sur chacun de ses éléments
   — retourne la **somme** des valeurs retournées par chacun des appels à `action_cb`
2. Remarquez que notre fonction `list_print` n'a plus d'intérêt : il suffit d'utiliser `list_apply` ! Mettez donc `list_print` en commentaire, modifiez le programme de test `tests/test_list_song.c` pour utiliser `list_apply` au lieu de `list_print`, testez.
3. Ecrivez une autre fonction-callback de votre choix dans le module song, et utilisez là dans le programme de test.

.../...

Exercice 8 Test des fuites mémoire avec valgrind

Relancez le programme de test (`tests/test_list_song.c`) dans l'utilitaire Valgrind.

Que constatez-vous ? Quel est le problème ?

Exercice 9 Gestion correcte de la libération de la mémoire

Vous l'aurez remarqué, quand on supprime un maillon de la liste générique, il faut être capable de libérer également la mémoire occupée par l'élément qui était stocké dans ce maillon.

Autrement dit, dans le cas où la liste contient des chansons, il faut être capable d'exécuter la fonction `song_delete` à chaque fois qu'on libère un mailon de la liste.

Mais bien sûr si la liste contient des hippopotames et non pas des chansons, c'est la fonction de libération des hippopotames qu'il faut exécuter, et non pas la fonction de libération des chansons...

En vous appuyant sur ce que vous venez d'implanter pour l'affichage des éléments d'une liste :

1. Dans le module song, définissez la fonction-callback `int song_delete_callback( void *ptr )`.
2. Dans le module list, modifiez le prototype et le code des fonctions `list_del_first` et de la fonction `list_delete` pour qu'elle soit capable de libérer correctement la mémoire allouée pour les éléments de la liste.
3. Modifiez le programme de test (`tests/test_list_song.c`) en conséquence.
4. Relancez le programme dans Valgrind et vérifiez qu'il n'y a plus de fuite mémoire.

Exercice 10 Chargement d'un fichier chanson dans une liste chaînée générique

Maintenant que nos listes génériques sont propres, intéressons nous à nouveau au module song_file.

1. Dans le module song_file, ajoutez la fonction `list_t song_file_load( char *filename, int *pNvalid, int *pNinvalid )` qui charge le fichier chanson nommé *filename* en mémoire dans une liste de chanson, stocke le nombre de chansons valides et invalides dans les paramètres pointeurs si ils ne valent pas `NULL` et retourne la liste.

   Remarquez que cette fonction et la fonction `int song_file_count( char *filename, int *pNvalid, int *pNinvalid )` précédemment écrite partagent le code qui lit une ligne et parse une chanson dans le fichier.

   Pour le factoriser ce code, vous écrirez et utiliserez la fonction `song_t library_read_next_song (FILE *stream, int *p_nbSkiped)`. Cette fonction lit retourne la prochaine chanson *valide* du fichier (compte tenu de l'endroit où l'on se situe dans le fichier), ou `NULL` si la fin du fichier est atteinte. Elle stocke de plus le nombre de chansons invalides (sautées) dans `*p_nbSkiped`.
2. Dans un nouveau fichier de test (`tests/test_song_file_load.c`), chargez un fichier librairie de chansons en mémoire sous forme de liste de chansons, puis affichez la liste et libérez-la.

   Ce programme prendra en paramètre (dans le Terminal) le nom du fichier à charger.

.../...

Exercice 11 Recherche par critère dans une liste générique

Nous voulons maintenant être capables de rechercher une chanson dans la liste une chanson, soit par son nom, soit par son identifiant, soit par le nom de son auteur.

Pour ce faire, il nous faudra parcourir la liste et pour, chacun de ses éléments, tester s'il satisfait le critère de recherche.

Il s'agira d'appliquer une fonction-callback à chaque élément de la liste, comme pour le pattern visiteur. Par contre, cette fonction-callback n'a plus pour rôle d'appliquer une simple action à l'élément, mais de si tester le critère de recherche est validé et de retourner un booléen.

En programmation, on peut appeler une telle fonction un *prédicat*.

De plus, cette fonction-callback prédicat doit cette fois ci prendre 2 paramètres, et non plus un seul : l'élément à comparer, comme précédemment, mais aussi un second paramètre qui est la valeur avec laquelle faire la comparaison (l'identifiant de la chanson recherchée, ou le nom de la chanson recherchée, etc. suivant le critère de recherche. Le type de fontion-callback `callback_t` n'est donc plus approprié. Il faut définir un nouveau type de fonction-callback, à deux paramètres.

1. Dans le fichier `callback.h` définissez le type `predicate_callback_t` ainsi :
   `typedef int (*predicate_callback_t)(void*,void*);`.
2. Dans le module song, définissez la fonction `int song_has_id_cb(void* ptr_elt, void* param)`. Cette fonction suppose que `ptr_elt` pointe en fait une chanson et que `param` pointe un entier qui représente un indice de chanson. Puis elle retourne 1 (`vrai`) si la chanson a l'indice voulu, 0 (`false`) sinon.
3. Dans le module list, écrivez la fonction
   `list_t list_search_next( list_t list, predicate_callback_t predicate_cb, void *param );`.
   Cette fonction retourne le premier élément de la liste pour lequel la maillon satisfait le prédicat `predicate_cb`, paramétré par `param`, ou une liste vide si aucun des éléments ne satisfait le prédicat.
4. Dans un nouveau programme de test `test_list_search`, chargez la librairie de chanson en mémoire puis affichez la chanson d'identifiant 2751 et vérifiez que la librairie ne contient aucune chanson d'identifiant 87543.

Maintenant que ceci est en place :

1. Ajoutez dans le module song une autre fonction prédicat, `song_is_by_cb`, qui teste si la chanson passée en paramètre est interprétée par l'artiste passé en paramètre.
2. Recherchez dans la librairie de chanson la première chanson interprétée par *Pink Floyd*.

Not So Difficult, is it ?

Exercice 12 Extraction d'une sous-liste chaînée par critère

Nous allons extraire de notre librairie de chansons la sous-liste des chansons satisfaisant un critère - par exemple toutes les chansons de *The Beatles*.

1. Dans le module list, écrivez la fonction
   `list_t  list_extract( list_t list, predicate_t predicate_cb, void *param )` qui retourne une *nouvelle* liste chaînée contenant tous les éléments de la liste d'origine qui satisfont le prédicat `prediate_cb`, paramétré par `param`.
   La liste initiale n'est, bien sûr, pas modifiée.
2. Complétez le programme de test `test_list_search` ainsi :
   — Extrayez de la librairie de chanson toutes les chansons interprétées par *Pink Floyd* et affichez ces chansons.
   — Libérez les deux listes chaînées (la librairie complète, et la liste des chansons de *Pink Floyd*.
   — Vérifiez avec Valgrind qu'il n'y a pas de problème d'accès mémoire ni de fuite mémoire.