Rendu TP : TP1 SDD
Agenda

BALLEJOS Lilian
LEGER Bertrand
Isima INP
Année Universitaire 2021 - 2022

Enseignant TP : CROMBEZ Loic
Date du rendu : 7 mars 2022

INP ISIMA
1 Rue de la Chebarde, 63178 Aubière
Tel : 04 73 40 50 00
Site web : isima.fr

Sommaire

1 Présentation du TP

Ce TP a pour but d'implémenter une liste simplement chainée qui représente un pseudo Agenda. Nous avons ici séparé notre structure de donnée en 3 sous structures :

- La structure **"tête"** qui est la tête fictive qui pointe sur la première semaine,
- La structure **"semaine"** qui contient un numéro de semaine,une année (toutes concaténées dans la même chaine de caractère), un pointeur sur la semaine suivante ainsi qu'un pointeur sur la liste des actions qui ont lieu cette semaine,
- La structure **"action"** qui possède un numéro correspondant à un jour de la semaine, une heure (le tout concaténées dans la même chaine de caractère), le nom de l'action et enfin un pointeur vers l'action suivante.

Nous avons créé un fichier source ".*c*" par structure (tete.c, action.c, semaine.c), un fichier "commun.c" qui contient toutes les fonctions en lien avec l'ensemble des 3 structures et enfin un fichier main.c qui contient le main.

Tous les fichiers sources sont stockés dans le dossier "source", les fichiers "headers" dans le dossier header, les fichiers objets dans le dossier "objet" et enfin les entrées et sorties de notre programme dans les dossiers "entree" et "sauvegarde". Un makefile est fourni pour tout compiler et génére l'executable **"Programme"**.

2 Présentation de la structure

2.1 Schéma de la structure

Voici un schéma de la structure que nous avons mis en place :

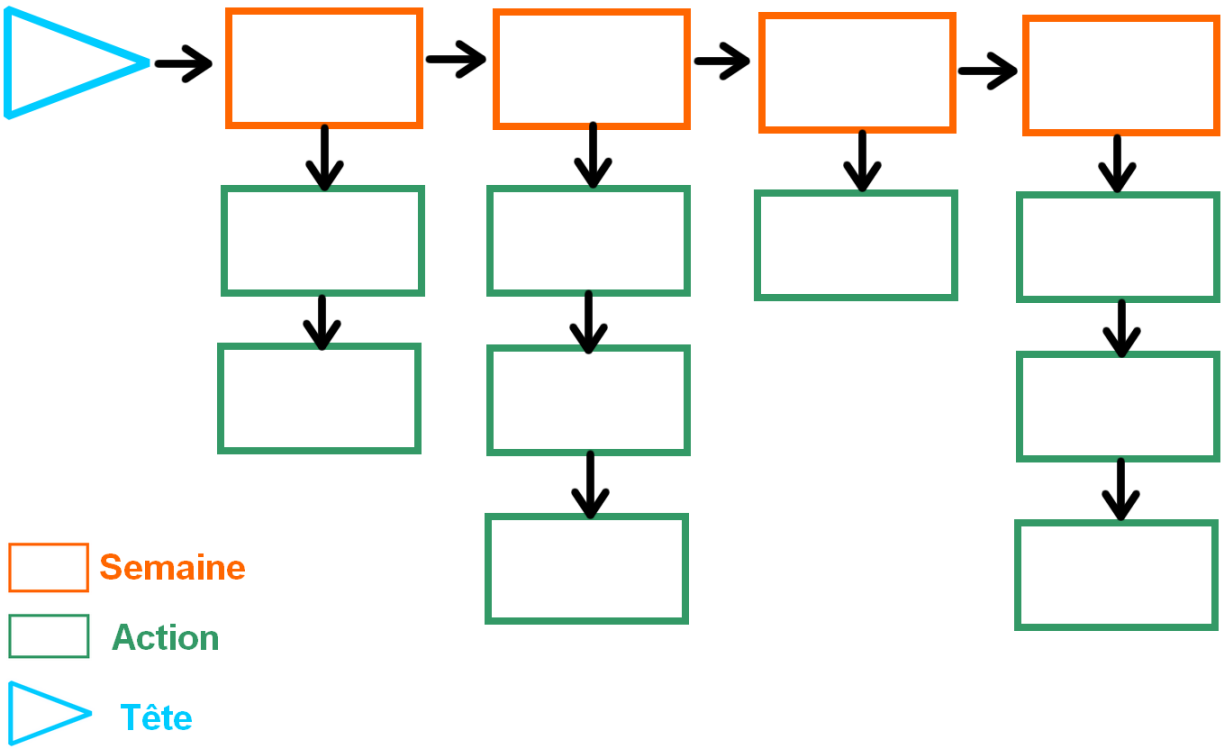

FIGURE 1 – Schéma de notre SDD "Agenda"

2.2 Format des fichiers en entrée et en sortie

Les formats des fichiers d'entrée et de sortie pour notre programme sont exactement les mêmes. Chaque ligne correspond à une activité et contient précisément dans cette ordre ces informations :

- une année sur 4 bytes
- un numéro de semaine sur 2 bytes
- un chiffre compris entre 1 et 7 correspondant à un jour (lundi, mardi ...)
- une heure sur 2 bytes (08 pour 8h par exemple)
- Un nom pour l'activité noté précisément sur 10 bytes (on complète avec des espaces si besoin)

Regardons maintenant un exemple de fichier d'entrée de notre programme et son fichier de sortie correspondant.

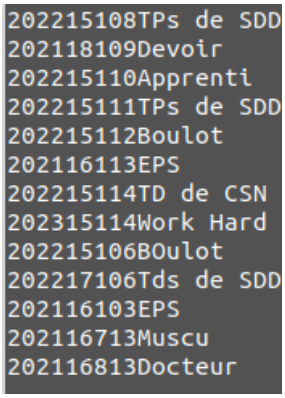

FIGURE 2 – Notre fichier d'entrée

Comme dit avant les informations sont bien notées dans le bon ordre. Une chose tout de même à remarquer ici : dans cette exemple nous avons fait exprès de glisser une erreur dans la ligne **"Docteur"** : en effet le chiffre des jours qui est censé être compris entre 1 et 7 vaut ici 8 !

Regardons maintenant les fichiers de sortie obtenu :

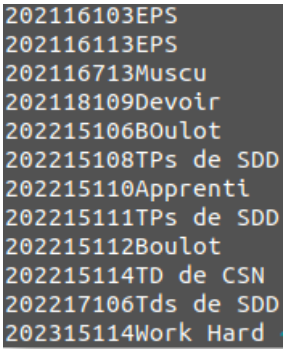

FIGURE 3 – Notre fichier de sortie

Après le passage dans notre programme les informations stockées dans notre fichier d'entrée on était triées par ordre croissant de leur date dans notre SDD ! Ainsi, à la sortie, toutes nos informations sont écrites dans l'ordre croissant dans notre fichier de sortie ! De plus on remarque que la valeur de **"Docteur"** étant mauvaise, notre programme ne l'a pas ajouté dans la SDD et ainsi elle n'est pas dans le fichier de sortie !.

3 Présentation des Fonctions

Passons maintenant à la présentation de toutes nos fonctions.

Mais avons cela nous nous devons de faire une aparté : nous avons pris la décision dans ce TP, dans chaque structure contenant des chaines de caractère de rajouter le caractère **'\0'** en fin de chaine. En effet ce caractère est un élément essentiel du langage C est nous n'avons pas voulu nous en séparer. Sinon nous aurions été obligé de recoder toutes les fonctions de **string.h**. Ainsi dans chaque structure les chaines de caractères sont sur un bit de plus afin de stocker ce dit caractère !

3.1 tete.h et tete.c

Le fichier **tete.h** possède la structure représentant notre tête fictive. Ainsi elle est seulement composée d'un pointeur pointant sur une structure de type "semaine".

```c
typedef struct tete{
    semaine * debut;
} tete;

```

FIGURE 4 – Structure "tete"

La seule fonction unique à la structure "tete" est **AllouеTete** qui comme son nom l'indique va allouer de la mémoire à notre tête fictive. La fonction prend en argument un pointeur indirect de type "tete" et renvoie 0 si tout va bien ou -1 si l'allocation a raté.

3.2 semaine.h et semaine.c

Passons à **semaine.h** qui contient la structure "semaine".

```c
typedef struct semaine{
    char annee_semaine[7];
    struct action * liste_action;
    struct semaine * suivant;
} semaine;

```

FIGURE 5 – Structure "semaine"

Comme avec "tete", la seule fonction en lien avec cette structure est **AllouеSemaine"** qui a pour but d'allouer de la mémoire à un élément de type "semaine". Cette fonction prend en argument un pointeur indirect sur une structure de type semaine, une string contenant une année et une autre string avec un numéros de semaine. Elle va tout de même contrôler si jamais les infos qu'ont lui a donné en paramètre sont valides (numéro de semaine bien compris entre 1 et 53). Elle renvoie 0 si tout c'est bien passé et -1 si l'allocation a raté ou les valeurs entrées sont mauvaises. L'année ainsi que le numéro de semaine sont concaténés dans le champs **annee_semaine**, cela nous evitera des tests en trop dans de futurs fonctions.

3.3 action.h et action.c

Enfin notre dernière structure est la structure **"action"** dont voici la forme :

```c
typedef struct action{
    char jour_heure[4];
    char nom[11];
    struct action * suivant;
} action;

```

FIGURE 6 – Structure "action"

La fonction en lien avec celle-ci est **AllouеAction** qui va allouer de la mémoire pour une case action. Elle prend en argument un pointeur indirect sur une action ainsi que 3 strings possédants respectivement, une valeur entre 1 et 7 pour le jour, une heure et un nom. La fonction va vérifier si le jour ainsi que l'heure est valide ! Si tout c'est bien passé (allocation réussi et champs valides) la fonction renvoi 0 sinon elle renvoi -1.

3.4 commun.h et commun.c

Passons à toutes les fonctions ayant un impact sur toute notre SDD !

3.4.1 Lecture et Ecriture dans les fichiers

Commençons avec **les gestions d'entrée et sortie :**

La fonction **"LectureFichier"** va tenter d'ouvrir le fichier en lien avec le nom qu'il reçoit en paramètre. Si cela est possible il va lire tout son contenu et stocker dans différentes strings les informations qu'il en tire. Une fois à la fin d'une ligne, on oublie pas d'enlever le caractère '\n' afin de passer à la ligne suivante puis on appelle la fonction **AjoutStruct** en lui donnant toutes les informations que l'on vient de récupérer. Une fois cela fait, on ferme le fichier. La fonction renvoie 1 si la lecture c'est bien passé, 0 sinon.

La fonction **"EcritureFichier"** va elle prendre en argument un nom de fichier et la tête fictive de notre SDD. Si on a pas de problème d'ouverture de fichier, on va parcourir entièrement notre SDD. C'est à dire semaine après semaine tout en parcourant toutes les actions de chaque semaine. Elle va, à chaque action, rajouter une ligne dans notre fichier avec les informations correspondant à celle-ci. On fait bien attention de rajouter un '\n' à chaque fin de ligne sauf la dernière pour respecter le format standard de nos fichiers. Grâce à cela un fichier produit par notre programme pourra être bien évidemment mis ensuite en entrée !

3.4.2 Gestion d'allocation

Voyons maintenant toutes les fonctions utilisant des allocations ou des libérations de mémoire !

La fonction **"CreationElement"** est une fonction outil qui prend en argument 5 strings avec toutes les informations possibles d'une activité (année, numéro de semaine, jour, heure, nom) ainsi qu'un pointeur indirect sur une semaine. Il va allouer en mémoire la semaine ainsi que l'action correspondant aux informations données et les relier entre eux ! La fonction renvoie 1 si tous c'est bien passé, 0 si une allocation a raté et -1 si les 2 ont ratés !

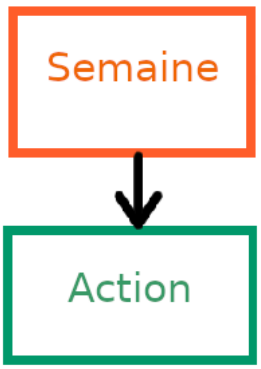

FIGURE 7 – Représentation de l'élément crée

La fonction **"LiberationElement"** fais l'inverse, elle libère un élément de la forme semaine pointant vers action. Le duo de ces des fonctions va nous faciliter la tâche lors de l'utilisation

de la fonction **"SupprStruct"**.

La fonction **"LibererStruct"** va entièrement libérer toute notre SDD. On lui donne simplement en argument notre tête fictive. Elle va parcourir l'intégralité de la SDD et libérer au fur et à mesure toutes les actions et les semaines. AU final on oublie pas de libérer la tête fictive !

3.4.3 Ajout et Suppression

On arrive aux deux fonctions les plus complexes. Elles sont intégralement commentées dans notre fichier "commun.c" et nous allons brièvement les expliquer ici !

La fonction **"AjoutStruct"** va être appelée par la fonction "LectureFichier" qui lui fourni toutes les informations sur une ligne de fichier ! Elle va ensuite crée un élément de la forme de la figure 7. Si on rencontre un problème lors de la création de cet élément, on le libère et on ne l'ajoute pas à la SDD ! Nous avons ensuite différents cas d'ajout à gérer :

- L'ajout quand la SDD est vide : on fait simplement pointer notre tête fictive sur le nouvel élément.
- L'ajout quand la case semaine existe déjà ! A ce moment là, on doit vérifier si l'action que l'on souhaite ajouter ne doit pas être ajouté en tête de la liste des actions de cette semaines. Si c'est le cas on modifie le pointeur "liste_action" de la dite semaine ! Sinon on fait un simple ajout avec les valeurs du pointeur précédent action ! On oublie pas de libérer la case semaine fraichement car une semblable existe déjà dans la SDD !
- L'ajout quand la semaine n'existe pas ! A ce moment là on doit vérifier que la semaine à ajouter n'est pas à positionner en tête de SDD, si c'est le cas on change le pointeur de notre tête fictive, sinon on ajoute simplement notre semaine avec le pointeur précédent semaine !

La fonction **"SupprStruct"** prend elle en argument un pointeur sur notre tête fictive et sur un élément de la forme de la figure 7 (c'est à dire une semaine pointant sur une action). On va parcourir toutes notre SDD en cherchant une semaine puis une action identique à celles prisent en argument ! Une fois cela fait, si on cette action existe, on la supprime, sinon on fait rien ! Si l'action est bien supprimée on renvoie 1, si la semaine visée n'existe pas dans la SDD on renvoie 0 enfin si l'action n'existe pas on renvoie -1. Nous avons encore une fois plusieurs cas à traiter :

- La semaine n'existe pas : on fait rien
- L'action n'existe pas : on fait rien
- On supprime la première action d'une semaine. A ce moment là on doit changer l'action pointé par la semaine puis libérer la mémoire de cette action.
- On supprime une action pas en tête. A ce moment là on juste a changer les pointeurs en faisant pointer le précédents de cette dite action sur la suivant de celle-ci puis de libérer la mémoire de cette action !
- Après suppression d'une action, on se rend compte que la semaine est maintenant vide ! On doit donc la supprimer elle aussi ! Soit c'est la première et on change le pointeur de la tête fictive soit on joue rediriger le pointeur précédent. Dans tous les cas on oublie pas de libérer la semaine à supprimer !

3.4.4 Affichage

La fonction **AffichageAgenda** est toutes simples. Elle parcourt entièrement notre SDD et affiche pour chaque élément "semaine" toutes les actions qu'il contient ! On utilise un tableau

statique contenant tous les jours de la semaine pour faire un affichage plus propre. Le principe est simple : On récupère la valeur ascii du jour à afficher auquel on enlève la valeur ascii du char "1". Ainsi, pour Lundi on aura l'indice 0, mardi l'indice 1 ... On a plus qu'à afficher la string présente à l'indice du tableau voulu !

3.5 contigue.h et contigue.c

Enfin voyons les fonctions en lien avec la liste contiguë qui cherche un motif. Nous rappelons que l'on doit, à l'aide d'un motif pris en argument, cherche si celui-ci apparait dans le nom de nos actions et si c'est le cas, stocker la date de cette actions dans une liste contiguë.
La liste contiguë est toutes simples : c'est un tableau de char en 2 dimensions statique ayant un nombre de ligne limité défini par la valeur **MAX_TAILLE_CONTIGUE** présent dans contigue.h. Chaque string contenu dans cette liste est de taille **TAILLE_DATE_ACTION**. Cette taille est de 10 car l'on stocke les 6 bytes de l'année ainsi que du numéro de semaine puis les 3 bytes du jour et de l'heure et enfin le char '\0'. Nous avons mis la liste contiguë en variable globale dans le fichier contigue.h. Ainsi on commence à définir deux pointeurs de type "char", un de début et un de fin et on appelle la fonction InitContigue qui va les placer correctement.

La fonction **InitContigue** prend donc en argument notre liste contiguë et deux pointeurs indirects "debut_contigue" et "fin_contigue". Au départ la liste étant vide, on les place tous les deux au début de la liste c'est à dire qu'ils prennent l'adresse "contigue[0]". Après cela on aura utilisera plus la notation "[ ]" des tableaux en C mais bien juste nos 2 pointeurs de début et de fin.

Ensuite, la fonction **RemplisContigue** prend en argument un motif, notre tête fictive et nos pointeurs de début et de fin. On va créer un pointeur "parcours_contigue" qui aura pour but de parcourir la liste contiguë et un pointeur "dernier_element" qui pointe sur la dernière position possible où écrire dans notre liste contiguë. Si on essaye d'écrire plus loin on n'écrit plus dans la zone fixe allouée à notre liste. C'est donc pour cela que l'on vérifie dans le if si on a dépassé cette valeur ou pas ! On a donc maintenant plus qu'à parcourir entièrement toutes notre SDD et vérifier si le motif apparait dans le nom des actions (et que la liste contiguë n'est pas pleine), si c'est le cas alors on ajoute les informations de date dans la liste. La fonction va ensuite en toute fin placer le pointeur de fin de la liste contiguë qui correspond à l'adresse où se trouve la dernière chaine de caractère de taille 10.

Au final **AfficheContigue** parcours tous les éléments existants dans la liste contiguë jusqu'à arriver au pointeur fin. On a utilisé une boucle for pour respecter les bonnes normes de code. Étant donné qu'on connait la taille de notre liste on passe par une boucle for ! On utilise le même tableau "tab_jour" que dans "AfficheStruct" pour afficher plus proprement les jours de la semaine.

4 Liste des cas

Nous allons ici vous présenter la liste des cas possibles et leurs jeux de donnée attitrés afin que vous puissiez tous les tester.

4.1 Lecture

4.2 Ajout

4.3 Suppression

4.4 Contiguë

5 Annexe

Table des figures