Programmation Impérative – TP 5
Tickets de caisse de la cafétéria

Guillaume Mercier
mercier@enseirb-matmeca.fr
Augusta Mukam
augusta.mukam@u-bordeaux.fr
Philippe Swartvagher
philippe.swartvagher@enseirb-matmeca.fr

2023 – 2024

On va développer un ensemble de fonctions qui permet de générer les tickets de caisse pour les clients d'une cafétéria.

Ce TP consiste à implémenter deux versions possibles de cette interface. À la fin du TP, vous rendrez les fichiers C correspondant à vos différentes implémentations sur Thor. Ensuite, vos implémentations seront testées automatiquement pour évaluer ce qui fonctionne et ce qui ne fonctionne pas. Par conséquent, **vous ne pouvez pas modifier l'interface** (les prototypes des fonctions : leur noms, les paramètres qu'elles prennent et leur type de retour ; en résumé : ne touchez pas aux fichiers `.h`), **ni le nom des fichiers**.

Récupérez le code de base.

À la fin de la séance, il faudra déposer sur Thor, dans le projet PG109 - Programmation Impérative, tous vos fichiers sources (`.c` et `.h`) en les mettant dans une archive `.tar.gz` :

```
tar cvzf archive.tar.gz cafeteria.c ticket.h ticket_item.h
   ticket_tab.c ticket_list.c
```

Le TP peut sembler long, mais ne vous inquiétez pas : je n'attends pas de vous que vous le terminiez.

1 Première implémentation : avec un tableau

Copiez le fichier `ticket.c` en `ticket_tab.c`. C'est dans `ticket_tab.c` que vous allez implémenter une première version des fonctions de l'interface, utilisant un tableau pour stocker les différents éléments achetés. Ce tableau aura *par défaut* une taille maximale de 10.

Complétez dans `ticket_tab.c` :

- le contenu de la structure `struct ticket_s` ;

- le contenu de la fonction `ticket_nouveau()` ;
- le contenu de la fonction `ticket_detruire()` ;
- le contenu de la fonction `ticket_ajoute_item()` ;
- le contenu de la fonction `ticket_afficher()` ;

Bien sûr, vous vous ferez un petit programme `cafeteria.c` pour tester au fur et à mesure que votre implémentation fonctionne. Pensez à tester les cas limites (pour vous assurer que vous gérez correctement les erreurs).

Compilez chaque fichier source séparément :

```
cc -Wall -Werror -c ticket_tab.c -o ticket_tab.o
cc -Wall -Werror -c cafeteria.c -o cafeteria.o
cc -Wall -Werror cafeteria.o ticket_tab.o -o cafeteria_tab
```

Dans un premier temps, le code suivant dans `cafeteria.c` :

```
struct ticket_s* ticket = ticket_nouveau();

ticket_ajoute_item(ticket, ENTREE, "Carottes râpées", 0.90);
ticket_ajoute_item(ticket, PLAT, "Steak", 3.55);
ticket_ajoute_item(ticket, PLAT, "Frites", 2.10);
ticket_ajoute_item(ticket, FROMAGE, "Camembert", 0.70);
ticket_ajoute_item(ticket, DESSERT, "Pomme", 0.70);

ticket_afficher(ticket, ticket_output);

ticket_detruire(ticket);
```

doit afficher la sortie suivante :

```
Entrée  Carottes râpées     0.90
Plat   Steak     3.55
Plat   Frites      2.10
Fromage Camembert     0.70
Dessert Pomme     0.70
5 items
TOTAL:   7.95 €
```

2 Seconde implémentation : avec une liste chaînée

On a un problème : certains clients achètent plus de dix éléments ! En réalité, on ne peut pas savoir à l'avance le nombre d'éléments sur chaque ticket et on ne veut pas être limité.

Dans cette seconde implémentation, on va utiliser comme structure de données une *liste chaînée* pour stocker tous les éléments d'un ticket.

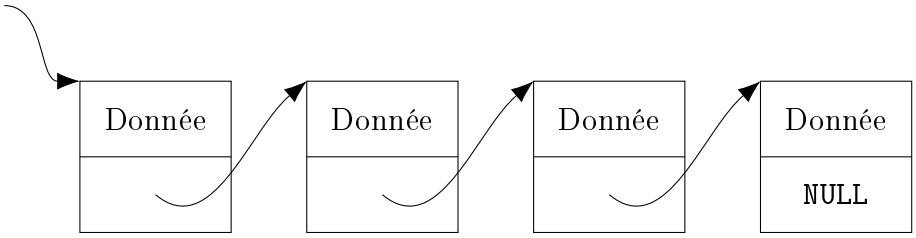

FIGURE 1 – Exemple de liste chaînée avec 4 éléments.

Présentation des listes chaînées

Chaque élément d'une liste chaînée possède deux membres :

- la donnée que contient cet élément ;
- l'adresse mémoire du prochain élément de la liste. S'il n'y a pas de prochain élément, on stocke la valeur `NULL`.

Une structure parente (au sens général : il peut en effet s'agir d'un `struct`, mais aussi d'un `void*`) stocke un pointeur sur le premier élément de la chaîne, pour désigner la chaîne tout entière. S'il n'y a pas d'élément dans la chaîne, le pointeur stocké est `NULL`. La Figure 1 représente une liste chaînée avec 4 éléments. Les différents éléments constituant la liste chaînée ne sont pas forcément stockés de façon contiguë en mémoire, d'où la nécessité de stocker à chaque fois un pointeur sur l'élément suivant.

Les opérations de manipulation d'une liste chaînée (ajout d'un élément, suppression d'un élément, parcours d'une liste) reviennent alors principalement à manipuler un pointeur sur l'élément suivant.

Avec cette structure de données, on peut stocker n'importe quel nombre d'éléments, sans connaître à l'avance ce nombre.

Implémentation

Copiez le fichier `ticket.c` en `ticket_list.c` et implémentez les mêmes fonctions que dans `ticket_tab.c`, mais en utilisant une liste chaînée.

Sans modifier `cafeteria.c` et sans recompiler `cafeteria.c` en `cafeteria.o`, vous devez pouvoir obtenir un programme `cafeteria_list` qui utilise l'implémentation avec les listes chaînées :

```
cc -Wall -Werror -c ticket_list.c -o ticket_list.o
cc -Wall -Werror cafeteria.o ticket_list.o -o cafeteria_list
```

`cafeteria_tab` et `cafeteria_list` doivent se comporter de la même manière (sauf dans le cas où vous ajoutez plus de dix éléments au ticket).

3 Changer la taille du tableau

Adaptez `ticket_tab.c` pour qu'on puisse préciser la taille du tableau qui stocke les éléments d'un ticket à la compilation, à l'aide de la constante de préprocesseur

`MAX_NB_TICKET_ITEMS`. Si la constante de préprocesseur n'est pas définie, la taille par défaut du tableau reste dix.

Exemples de compilations :

```
cc -Wall -Werror -c ticket_tab.c -o ticket_tab.o # limité à
   10 items
cc -Wall -Werror -DMAX_NB_TICKET_ITEMS=25 -c ticket_tab.c -o
   ticket_big_tab.o # limité à 25 items
```

4 Imprimer le ticket

On passe un descripteur de fichier de type `FILE*` à la fonction `ticket_afficher()` et c'est sur ce descripteur de fichier que le ticket est écrit.

Modifiez votre programme `cafeteria.c` pour qu'il accepte comme argument un nom de fichier dans lequel écrire le ticket. Si le fichier n'existe pas, il sera créé. Si le fichier existe déjà, le ticket sera écrit à la suite du contenu déjà présent dans le fichier. Si aucun argument n'est fourni, on écrira sur `stdout` (la sortie standard).

5 Supprimer un élément du ticket

Oups, on a ajouté le mauvais élément sur le ticket ! Vite, il faut le supprimer !

Implémentez maintenant la fonction `ticket_supprime_item()` dans `ticket_tab.c` et `ticket_list.c`. Cette fonction supprime le $i^{\text{ème}}$ élément du ticket (en commençant à compter à partir de zéro : pour supprimer le premier élément, on utilisera $i = 0$).

6 Bonus

- Implémentez la fonction `ticket_somme_tous()` qui renvoie la somme des montants de tous les tickets qui ont été affichés.
- Modifiez l'affichage généré par `ticket_afficher()` pour grouper par type les différents éléments. Par exemple :

```
Entrée
Carottes râpées     0.90

Plat
Steak     3.55
Frites      2.10

Fromage
Camembert     0.70

Dessert
Pomme     0.70
```

```
5 items
TOTAL:  7.95 €
```

- Toujours sans modifier le prototype de la fonction `ticket_ajoute_item()`, détectez (lors de l'appel à `ticket_ajoute_item()` ou `ticket_afficher()`, à vous de choisir) si plusieurs éléments ont été ajoutés plusieurs fois. Si c'est le cas, adaptez l'affichage ainsi :

```
Frites x2   2.10 x 2 = 4.20
```

- Rendez la présentation du ticket imprimé plus jolie, notamment, alignez les prix à droite :

```
     ** Entrée **
Carottes râpées     0.90

      ** Plat **
Steak               3.55
Frites              2.10

     ** Fromage **
Camembert           0.70

     ** Dessert **
Pomme               0.70

5 items
           TOTAL:  7.95 €
```