TRAVAUX PRATIQUES XII

Files par tableaux circulaires

L'objectif de se TP est d'implémenter le type abstrait des Files vu en cours, par la méthode des tableaux circulaires.

A Rappels et complémentrs sur les interfaces

A.1 Interface : le pourquoi du comment

Assez vite, les projets de programmation atteignent une taille telle qu'il ne tiennent plus dans un seul fichier : on ne s'y retrouve plus. Il est alors pertinent de les couper en fichier distincts, chacun ayant un rôle précis.

On obtient ainsi plusieurs `.c`, qu'il faut compiler ensemble. Mais chaque `.c` reste compliqué, sans parler du fait qu'il a pu être codé par quelqu'un d'autre [1]. Il est donc important, capital, d'avoir un « résumé » de chaque `.c` : c'est le rôle des headers `.h`. On parle aussi d'interface.

Dans une interface, on annonce l'existence et donne la documentation des fonctions du `.c` qui sont utilisables. Une autre façon de le dire est que l'interface correspond à un type abstrait (avec une documentation qui lui donne une sémantique), alors que le fichier source associé est une implémentation de ce type abstrait [2].

Un autre intérêt des interfaces est de permettre de cacher une partie du `.c` : toute fonction du code source qui n'apparait pas dans l'interface n'est pas utilisable par un code client [3]. Cela permet de rendre inaccessibles certaines fonctions qui sont « auxiliaires » (servent uniquement à réaliser une autre fonction plus intéressante).

A.2 Idempotence de l'inclusion

« Idempotence », ça claque !

Mettons que l'on ait un fichier `A` qui appelle un fichier `B` et un fichier `C`. Mettons également que `B` appelle également `C`.

Dans ce cas, compiler `A` va demander d'inclure `C` deux fois : une fois car `B` (qui est inclus) le demande, et une fois car `A` le demande lui-même. Si on ne fait pas attention, cela provoquerait un bug car l'éditeur de liens aurait l'impression que l'on définit deux fois toutes les fonctions de `C`. Or, il est interdit de définir deux fois le même identifiant.

Ce n'est pas un cas théorique abstrait : par exemple, `stdlib.h` et `stdint.h` appellent tous deux une même librairie `stddef.h`. Dès lors, lorsque l'on inclut `stdlib.h` et `stdint.h`, `stddef.h` est appelé deux fois…

Le problème ici est qu'a priori inclure deux fois un fichier n'a pas le même effet que l'inclure une seule fois. On dit que l'inclusion n'est pas *idempotente*. Nous pouvons cependant par une modification très simple la rendre idempotente.

Pour cela, nous allons faire en sorte qu'inclure le `.h` :

- Définisse une « variable » qui dise « ce fichier a déjà été inclu ».
- Et que si cette variable est déjà définie, rien ne se passe (on ne déclare alors aucune fonction, comme ça pas de doublon).

Notez que cette « variable » n'a besoin d'exister qu'à la compilation. Ce n'est pas une variable du code, c'est une variable « du compilateur ».

Cela est réalisé avec des `#define` et `#ifndef` comme ci-dessous :

1. Ou vous-même il y a longtemps, croyez-moi cela ressemble à quelqu'un d'autre.
2. Cette façon de voir atteint ses limites lorsque le `.c` n'implémente pas vraiment un type. Mais c'est la bonne façon de penser : cahier des charges abstrait *VS* réalisation concrète.
3. On dit qu'un fichier A est client de B s'il utilise B.

C
```c
/* << Si h n'est pas déjà définie >> */
#ifndef H

/* << Définir H (et lui donner la valeur 0) >> */
#define H 0

...
ici, mettre tout le header normal
....

/* << fin du Si >> */
#endif
```

Et voilà ! Grâce à cette astuce, l'inclusion est devenue idempotente. Seule la première inclusion fait quelque chose, les inclusions suivantes ne font rien car la « variable-compilateur » existe déjà et que l'on ne rentre donc pas dans le Si.

Rmq hors-programme Ce mécanisme qui permet de ne compiler des morceaux de code que sous certaines conditions est très utilisé dans les librairies standard C. On s'en sert par exemple pour créer des codes différents selon les spécificités de la machine sur laquelle on compile : l'OS détecte le matériel, définit en conséquence des « variables-compilateur » associé, et le compilateur peut alors compiler ou non des morceaux de code en fonction.

B Tableau circulaire

On travaille **en C**. Un zip contenant les fichiers à compléter est en ligne. Le `main` de `test.c` est légèrement pré-complété par du code commenté : ignorez cela pour l'instant, nous en reparlerons en question 3..
On rappelle que les options suivantes doivent être utilisées à la compilation :

```
-Wall -Wextra -fsanitize=undefined,address
```

Les sanitizers pourront être ôtés dans la version finale.
On rappelle que le TP précédent a défini la compilation séparée.

Dans cette section, on se propose d'implémenter les files **d'entiers** en les stockant dans un tableau circulaire. Pour cette représentation, on suppose que nos files auront toujours une longueur bornée : à la création de la file, on devra indiquer la capacité maximale attendue pour la file ; et la file ne pourra jamais en déborder.

Définition 1

Rappel : Dans un tableau circulaire ayant C cases, les indices sont considérés modulo C. Ainsi le tableau est indexé de 0 à C-1, mais la « case d'indice C » est également la case d'indice 0, « la case C+1 » est également la case d'indice 1, etc. Comme si le tableau bouclait sur lui-même.
La case qui suit la case d'indice `i` est donc la case d'indice `(i+1)%C`.

Pour implémenter une file dans un tableau circulaire, on utilise un type struct contenant :

- Un pointeur `tab` vers une zone de la mémoire contenant les cases du tableau circulaire.
- Un entier `capacite` qui contient le nombre de cases de la zone en question.
- Un indice `sortie` qui contient l'indice de l'extrémité (de la sortie) de la file.
- Un entier `len` qui contient le nombre d'éléments de la file, donc le nombre de cases utilisées.

Ainsi, les éléments de la file sont rangés dans les cases d'indices `sortie`, `sortie+1`, ..., `sortie+len-1` (attention cette énumération se fait en prenant en compte la cyclicité).
N'hésitez pas à faire des schéma en plus de ceux que nous avons fait en cours et au tableau. N'hésitez pas à me demander de l'aide.

Voici la structure correspondant à ce que l'on a annoncé :

C : file.h
```c
struct file_s {
  int* contenu;  // le tableau circulaire
  int capacite;  // le nb max d'elem de la file
  int sortie;    // l'indice du prochain élément à sortir
  int len;       // le nombre d'éléments dans la file
};
```

1. Cette structure définit un type nommé « `struct file_s` ». Créez un nouvel identifiant pour ce type, « `file` », à l'aide d'un `typedef`.

   Pour rappel, `typedef type new_id;` crée un nouvel identifiant « `new_id` » pour le type « `type` ».

2. Implémentez les différentes fonctions de l'interface. On pourra utilise des `assert` pour vérifier des préconditions (par exemple, on ne peut pas ajouter un élément à une file pleine ou en ôter à une file vide…).

   Testez vos fonctions !

   Vous n'êtes pas obligé·e·s de les faire dans l'ordre. À titre indicatif, je les corrigerai dans cet ordre :

   — `file_vide` .
   — Puis les 4 accesseurs.
   — Puis les 2 transformateurs.
   — Puis `file_vide` .

3. L'idée est maintenant d'implémenter l'exercice sur la queue de la cantine que nous avons fait en classe. Commencez par aller le relire.

   Nous allons travailler comme suit :

   — Vous écrirez dans un fichier texte (`.txt`) les mouvements de la cantine. Pour être précis, ce fichier devra contenir :
      — Sur sa première ligne, le nombre de mouvements (« le nombre de cases du tableau »).
      — Sur sa seconde ligne, séparés par des espaces, les mouvements (un entier positif si un élève entre dans la queue, négatif s'il sort).
   — La fonction `lecture` pré-codée permet de lire ce fichier, et d'obtenir le tableau associé. Cette fonction a besoin de connaitre le nom `filename` du fichier : en fait, on va le lire sur la ligne de commande !
   — Décommentez les morceaux commentés du `main`. Ils permettent de lire la ligne de commande qui a lancé le programme : `argc` contient le nombre de « mots » de la ligne de commandes (définis comme étant séparés par des espaces), et `argv` est un tableau qui contient dans l'ordre chacun de ces mots (qui sont donc de type `char*`).

     Par exemple, la ligne de commande « `./exe starlight.txt` » donne lieu à `argc` valant 2, et `argv` valant le tableau `[|"./exe"; "starlight.txt"|]`.

     Le code que vous avez décommenté permet justement de lire la ligne de commande, d'en déduire le fichier où sont écrits les mouvements, de le lire et d'obtenir `mouv` le tableau des mouvements.

   Il ne vous reste qu'à résoudre l'exercice.

4. Créez des fonctions permettant de convertir un tableau en file ou une file en tableau. Ajoutez-les au header.