Licence 2, Partiel LC4, Développement en C

17 mars 2007, durée 2h

Documents manuscrits et tous les documents de cours sont autorisés (tds, tps, corrections, cours). **Les livres ne sont pas autorisés.** *Tous les exercices sont indépendants. Il n'est pas demandé de faire* `#include` *pour les fichiers en tête de la bibliothèque standard.*

Votre code doit être écrit de façon lisible, avec des indentations et des accolades appropriées permettant de distinguer les fins de blocs de code (fins de boucles, ...).

Le barème est donné seulement à tire indicatif.

1 Question de cours

Exercice 1 — *1 point* Écrire un fichier `Makefile` qui permet de compiler un programme C composé des fichiers

- `fonctions.c` qui contient des définitions de fonctions mais pas la fonction `main`,
- le fichier en tête `fonctions.h` correspondant aux fonctions de ,fonctions.c
- le fichier `gestion.c` qui contient la fonction `main`. On sait que la fonction `main` utilise les fonctions définies dans `fonctions.c`. En particulier `gestion.c` fait inclusion de `fonctions.h`.

2 Pointeurs

Exercice 2 — *2.5 points* Écrire une fonction

```
int *concat(int* t1, int n1, int* t2, int n2)
```

qui prend en paramètre deux tableaux `t1` et `t2` d'éléments de type `int`. Les paramètres `n1` et `n2` donnent le nombre d'éléments de chaque tableau. La fonction doit retourner le tableau `t1` contenant la concaténation de tableaux `t1` et `t2`. On supposera que

- le tableau `t1` a été alloué auparavant par un `malloc`, donc vous pouvez (et devez) utiliser `realloc` pour changer sa taille,
- le tableau `t2` a été alloué aussi par `malloc` et si la fonction `concat` a réussi de créer la concaténation de `t1` et `t2` alors vous devez libérer la mémoire occupée par `t2`. Par contre si `concat` échoue ni `t1` ni `t2` ne devez pas changer.

Exercice 3 — *2 points* Indiquer ce qu'affichera le programme suivant :

```
#include <stdio.h>
int main(void)
{
  int a,b;
  int *p1,*p2,*p3, *pa, *pb;
  int tab[]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
  pa = tab;
  printf("E1 = %d\n", *(pa+2));
  pb = pa + sizeof tab / sizeof tab[0];
  printf("E2 = %d\n", *(pb-2) );
  pa  += 2;
  printf("E3 = %d\n", *pa);
  pa = &tab[8];
  printf("E4 = %d\n", *(pa-2) );
  pb = &tab[12];
  a = pb - pa + 1;
  printf("E5 = %d\n", a);
  a=8;
  b=2;
  p1=&a;
  p2=&b;
```

```
  p3=p2;
  a=++*p2 * *p3;
  printf("E6 = %d\n", a);
  b += *p1;
  printf("E7 = %d\n", b);
  return 1;
}
```

3 Boucles et décalages

Exercice 4 — *2.5 points* Écrivez une fonction

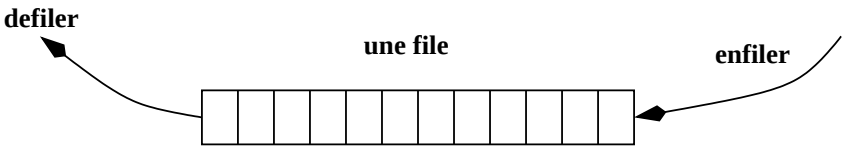
`int position(unsigned int x)`

qui renvoie la position du bit de poids le plus fort de `x` qui soit égal à 1. Cette fonction renverra -1 si `x = 0`.

4 Files

Les exercices qui suivent implémentent les opérations sur une file. Même si tous ces exercices forment un ensemble ils peuvent être écrits dans n'importe quel ordre. En particulier, si vous êtes bloqués sur un exercice n'hésitez pas à passer à l'exercice suivant.

Rappel : Une file est une structure de donnée dans laquelle les premiers éléments ajoutés sont les premiers à être récupérés (comme dans une file d'attente) :



Pour implémenter une file nous allons utiliser la structure file_t et le type file_t suivants :

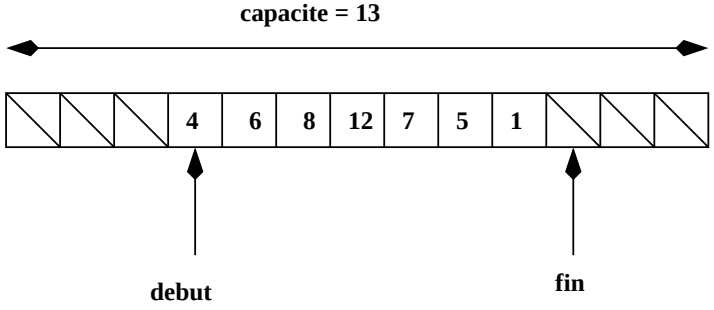
```
struct file_t {
    size_t debut, fin, capacite;
    int *elements;
};
typedef struct file_t file_t;
```

Une file d'entiers sera implémentée (cf. figure) en stockant tous les éléments de la file dans le tableau pointé par le champ elements. Le champ `capacite` donne le nombre d'éléments de tableau `elements`. Les champs `debut` et `fin` donnent les indices dans le tableau `elements` :

- `debut` donne l'indice de l'élément qui se trouve en tête de la file, c'est cet élément qui est le plus ancien sur la file et qui sera enlevé par la prochaine opération `defiler`,
- `fin` donne l'indice du premier élément libre dans le tableau `elements`.

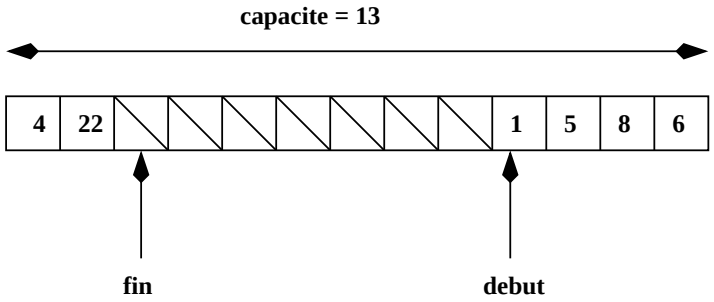
Les éléments de la file sont rangés dans les tableau `elements` de façon circulaire. Il y a trois cas possibles :

(1) `debut < fin` comme sur le dessin suivant :



Le dessin ci-dessus représente une file contenant `4,6,8,12,7,5,1`. L'élément 4 est le plus ancien sur la file et donc le premier à être défiler, l'élément 1 est le plus récent (le dernier qui a été enfilé).

(2) `fin < debut` comme sur le dessin suivant :



Le dessin ci-dessus représente une file contenant `1,5,8,6,4,22`, l'élément 1 étant le plus ancien sur la file (il est en tête) et 22 le plus récent,

(3) `debut == fin`. Dans ce cas la file est vide.

L'indice `fin` donne toujours une case libre, c'est-à-dire dans un tableau de `capacite` d'éléments nous pouvons stocker une file contenant d'au plus `(capacite - 1)` éléments.

Exercice 5 — *3 points* Écrire les fonctions

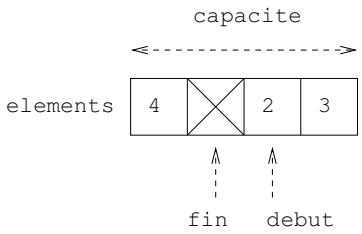
```
file_t *alloue_file(size_t capacite);
void libere_file(file_t *file);
```

- La fonction `alloue_file` alloue et initialise une file vide (initialisez `debut` et `fin` à 0). Le paramètre `capacite` donne le nombre de cases que devait avoir le tableau `elements`. Elle renvoie le pointeur vers la structure `file_t`.
  La fonction `alloue_file` renverra NULL si l'allocation mémoire échoue ou si capacite $<= 0$.
- La fonction `libere_file` libère l'espace mémoire occupé par une file et ses éléments.

Rappelons que la file est vide lorsque les indices de tableau debut et fin sont égaux.

Le nombre d'éléments de la file est égal au nombre de cases utilisées du tableau.

Le dessin ci-dessus montre une file contenant 3 éléments. Notez que cette file est pleine, pour ajouter un élément il faudra soit d'abord en enlever au moins un soit augmenter la taille de tableau `elements`.



Exercice 6 — *3 points* Écrire les fonctions

```
int est_vide(file_t *file);
size_t taille(file_t *file);
int est_pleine(file_t *file);
```

qui, respectivement,

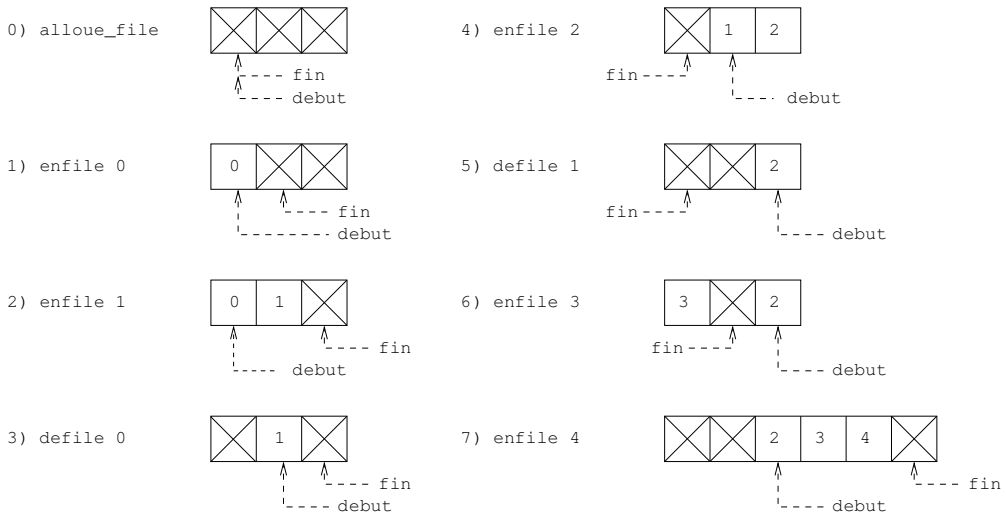
- `est_vide` renvoie 1 (ou une autre valeur différente de 0) si la file est vide et 0 dans le cas contraire,
- `taille` renvoie le nombre d'éléments dans la file,
- `est_pleine` renvoie 1 si la file est pleine et 0 sinon.

Dans la figure suivante, on montre comment les indices debut et fin ainsi que le tableau pointé par elements sont modifiés par différentes opérations successives sur une file. Les indices debut et fin sont autorisés à dépasser la fin du tableau pour revenir au début.

Lorsqu'on veut ajouter (enfiler) un élément à la fin de la file, on place l'élément dans la case indicée par fin et on décale l'indice de fin vers la droite. Les éléments de la file étant rangé de façon circulaire on suppose ici que « à droite »de l'indice `i` se trouve l'indice `(i+1) % capacite` (rappelons que pour deux entiers `m` et `n` l'expression `m % n` donne en C la valeur de `m` modulo `n`), c'est-à-dire « à droite »de l'indice `capacite-1` se trouve l'élément de l'indice 0.

Si on veut enfiler un élément et la file est pleine alors on redimensionne le tableau `elements` avec realloc () à deux fois sa taille initiale. Notez que si `fin < debut` alors un simple `realloc` ne suffit pas, il faudra déplacer les éléments de la file pour occuper les cases consécutives dans le tableau `elements`.

Lorsqu'on veut enlever (défiler) le début de la file, on décale l'indice de début vers la droite. Lors de ces opérations de décalage, on devra faire attention à toujours rester dans le tableau, quitte à passer de la dernière à la première case.



Exercice 7 — *6 points* Écrire les fonctions

```
int enfile(file_t *file, int n);
int defile(file_t *file);
```

- La fonction `enfile` enfile un entier n (la fonction renverra `n` si l'opération se termine avec succès et une valeur différente de `n` si l'éventuelle (ré)allocation mémoire a échoué),
- La fonction `defile` défile l'entier au début de la file. et renvoie sa valeur. Si l'utilisateur essaie de défiler d'une file vide alors le programme se terminera avec un message d'erreur (il suffit de mettre une assertion

  ```
  assert( ! file_vide(file) );
  ```

  au début du code de la fonction.)