Programmation C
TP n° 11 : Généricité

Exercice 1 : Pointeur Générique - Mise en jambes

Cet exercice est là pour s'assurer que l'on comprend ce que signifie un pointeur de type `void *`, appelé *pointeur générique*. Ouvrir un `main` et faites ce qui est demandé.

1. Définir une variable entière `a` puis un pointeur générique `pt` qui prend l'adresse de `a`
2. En une instruction n'utilisant que le pointeur `pt` et sans utiliser `a`, changer la valeur de `a` par 42
3. De façon similaire, élever `a` au carré en utilisant uniquement le pointeur `pt`
4. Définir le type `paire` suivant

```
typedef struct{
  int x,y;
}paire;
```

5. Définir une variable `b` de ce type `paire` et faire pointer `pt` vers `b`
6. Incrémenter le champ `y` de `b` juste en utilisant `pt`

Exercice 2 : Fifo Generique

Préambule : pour cet exercice, on respectera les principes de programmation modulaire : un fichier .c pour le code source, un header .h associé et un dernier fichier .c avec juste un main pour tester les fonctions. Ce n'est pas explicitement demandé, mais comme d'habitude, vous ferez au fur et à mesure un test dans votre main pour chacune des fonctions qu'on vous demande d'écrire.

Le but de cet exercice est d'implémenter une file (FIFO) générique, c'est à dire qu'on pourra se servir de cette structure pour des files de `int`, des files de `double`, ou n'importe quel autre type. Rappelons qu'une file est une suite d'éléments du même type telle que les éléments sont retirés de la file dans l'ordre d'arrivée (first in first out). La file `fifo` sera implémentée à l'aide d'un tableau qui contient les éléments de la file et d'une structure auxiliaire qui permet d'implémenter la gestion de la file. La structure auxiliaire et la file (fifo) sont définies comme :

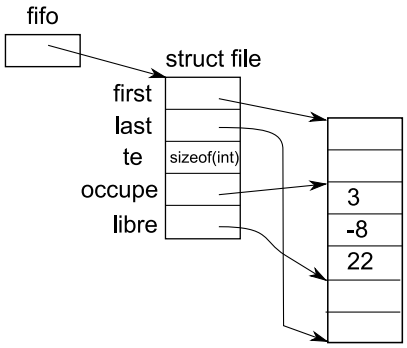
```
struct file{
  void *first;     /*pointeur debut de tableau*/
  void *last;      /*pointeur fin de tableau*/
  size_t te;       /*taille d'un element en octets*/
  void *occupe;    /*pointeur premier element de la file*/
  void *libre;     /*pointeur le premier element libre*/
};

typedef struct file *fifo;
```

La structure `struct file` possède les champs suivants :

- `first` est le pointeur vers le premier élément du tableau dans lequel on stockera les éléments de la file.
- `last` est le pointeur vers le premier octet juste **après** le dernier élément du tableau, ce pointeur sert à marquer la fin du tableau.
- `te` est la taille d'un élément en octets. Notre but est d'implémenter une pile générique, la taille d'un élément sera fixée à la création de la pile.
- `occupe` est l'adresse du premier élément dans la file, c'est l'adresse de l'élément qui sera retiré de la file par la prochaine opération `get`.
- `libre` le pointeur vers l'octet juste après le dernier élément de la file, la prochaine opération `put` mettra un nouveau élément à cette adresse.

La figure ci-contre représente une file de nombres entiers. Dans la file il y a actuellement trois éléments : $3, -8, 22$. Et la capacité maximale de la file est de 7 éléments. La prochaine opération `get` obtiendra le premier élément 3 qui sera retiré de la file. La prochaine opération `put` ajoutera un nouvel élément après 22.



A chaque moment les pointeurs utilisés doivent satisfaire la condition suivante :

```
first <= occupe <= libre <= last
```

(attention cependant, ce sont des pointeurs `void*` donc la comparaison ou l'arithmétique sur ces pointeurs est interdite, voir question 1 plus bas).
De cette description il découle que :

1. `occupe == libre` si et seulement si il n'y a aucun élément dans la file,
2. `(libre - occupe)` divisé par `te` donne le nombre d'éléments actuellement dans la file,
3. `(last - first)` divisé par `te` donne le nombre maximal d'éléments que nous pouvons stocker dans la file quand toutes les places sont occupées,
4. `libre == last` s'il n'y a plus de place pour le nouveau élément (l'adresse `libre` n'est plus une adresse valide d'un élément du tableau, il faudra réorganiser la file pour pouvoir insérer un nouveau élément).

1. On l'a dit, grâce à cette structure, on pourra créer des files d'entiers, des files de double, etc... C'est le champ `te` qui permettra de savoir la taille du type qu'on stocke. *Attention, pour permettre la généricité, les pointeurs vers les données sont de type `void *`, ce qui présente un inconvénient :* le C standard n'autorise pas l'arithmétique de pointeurs sur les void*, ce qui est a priori bien normal puisqu'on ne sait pas quel type est pointé. En réalité des compilateurs comme `gcc` acceptent ce genre d'opérations en considérant les `void*` comme des `char*` (et donc faire +1 correspond à avancer d'un octet). Pour ne pas reposer sur cela, et pour éviter de faire des cast vers `char*` en permanence, nous allons utiliser une fonction auxiliaire qui va nous permettre de faire de l'arithmétique sur des `void*`, cela nous sera utile par la suite. Ecrire donc une fonction

```
static void* decale(void* f, ssize_t d){
```

qui va retourner l'adresse `f` décalée de `d` octets, (c.a.d. `f+d` si `f` était un `char*`).

2. Écrire la fonction

```
fifo create_fifo( size_t capacite_init, size_t taille_elem )
```

qui crée une file vide (sans éléments), i.e. crée la structure et le tableau. Le paramètre `capacite_init` c'est la taille initiale du tableau qui stocke les éléments de la file (la taille mesurée en nombre d'éléments et non pas en octets). Le paramètre `taille_elem` est la taille d'un élément en octets. Par exemple une file de 7 `int` (comme celle sur la figure ci-dessus) sera créée par l'instruction `create_fifo( 7, sizeof(int) )`. Initialement les trois pointeurs seront égaux : `first == libre == occupe` et la fonction `create_fifo` retourne NULL si la création échoue (par exemple problème de `malloc`), sinon elle retournera un `fifo`, c'est-à-dire l'adresse d'un `struct file`.

3. Écrire la fonction suivante qui supprime la file

```
void delete_fifo(fifo f)
```

4. Écrire la fonction suivante qui retourne 1 si la file est vide et 0 sinon :

```
int empty_fifo(fifo f)
```

5. Écrire la fonction

```
void *get_fifo(fifo f, void *element)
```

qui récupère le premier élément de la file. La fonction doit copier cet élément à l'adresse `element` (pensez à utiliser la fonction `memmove`) et elle doit le supprimer de la file (ce qui revient à changer la valeur de pointeur `occupe`).
La fonction retourne NULL si la file est vide (et rien ne sera écrit à l'adresse `element`), sinon elle retourne `element`.

6. Écrire une fonction auxiliaire

```
static void *put_fifo_no_shift( fifo f, void *pelem )
```

qui met un élément dans la file uniquement quand il y a de la place à la fin de la file.
Le paramètre `pelem` contient l'adresse de l'élément qui sera recopié à l'adresse `libre`). Cette fonction permet d'ajouter un nouveau élément dans la file uniquement si `libre < last`, i.e. `libre` donne l'adresse à l'intérieur de la file. Dans ce cas la fonction retourne l'adresse du nouveau élément dans la file.
Si `libre == last` c'est-à-dire il n'y a plus de place à la fin de la file alors la fonction retournera `NULL` et le nouveau élément ne sera pas mis dans la file.

7. Écrire la fonction

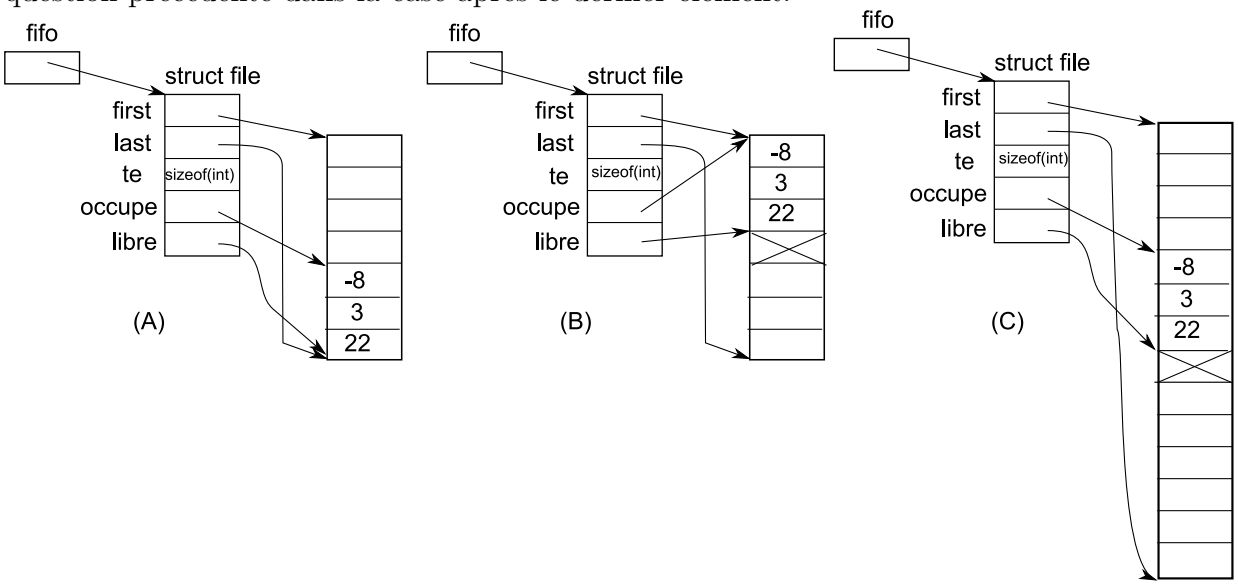
```
void *put_fifo(fifo f, void *pelem)
```

qui met un nouveau élément à la fin de la file. Contrairement à la fonction précédente, cette fonction doit insérer un élément même s'il n'y a plus de place après le dernier élément dans la file. S'il y a de la place à la fin de la file alors on procède comme dans l'exercice précédent. Supposons maintenant que `libre == last` c'est-à-dire il n'y a pas de place à la fin de la file, comme dans la figure (A) dessous.
Il y a alors deux algorithmes possibles pour insérer un nouveau élément dans la file dans cette situation.

(B) Le premier algorithme déplace (`memmove`) tous les éléments au début du tableau comme dans la figure (B) ci-dessus. Maintenant le nouveau éléments sera inséré dans la case marquée, juste après le dernier élément 22. Cet algorithme est applicable uniquement si le nombre de cases occupées est inférieur au nombre total de cases dans le tableau. Noter que, une fois le

déplacement des éléments et ajustement de pointeurs effectué, l'insertion peut se faire avec la fonction de la question précédente.

(C) Le deuxième algorithme agrandit le tableaux deux fois (`realloc`) ce qui donne des nouvelles cases libres après le dernier élément dans la file, voir la figure (C) ci-dessous. Une fois le tableau agrandi et les pointeurs ajustés, l'insertion peut se faire avec la fonction de la question précédente dans la case après le dernier élément.



Dans votre implémentation de la fonction `put_fifo`, s'il n'y a plus de place après le dernier élément, vous devez utiliser le deuxième algorithme quand le pourcentage de cases libres est inférieur à 25%, c'est-à-dire quand

```
(libre - occupe) < (last - first)*0.25
```

Dans le cas contraire vous devez appliquer le premier algorithme.