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

```c
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 sera implémentée à l'aide d'un espace de stockage qui contiendra les éléments de la file, et d'une structure auxiliaire qui permettra d'implémenter la gestion de la file. La structure auxiliaire est définie à l'aide du type suivant :

```c
struct fifo {
  void *first;     /* debut de l'espace de stockage */
  void *last;      /* fin de l'espace de stockage */
  size_t te;       /* taille d'un element en octets */
  void *occupe;    /* adresse du premier element de la file */
  void *libre;     /* adresse qui suit le dernier element */
};

typedef struct fifo fifo;
```

La structure `struct fifo` possède les champs suivants :

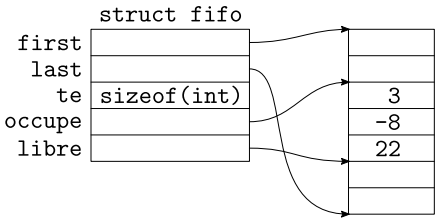
- `first` est l'adresse du premier octet de l'espace mémoire dans lequel on stockera les éléments de la file.
- `last` est l'adresse premier octet immédiatement **après** le dernier octet de cet espace.
- `te` est la taille d'un élément en octets - notre but étant d'implémenter une pile générique, la taille d'un élément sera fixée à la création de la pile.
- `occupe` est l'adresse du premier élément de la file, celui qui sera retiré de la file par la prochaine opération `get` (voir ci-dessous).
- `libre` est l'adresse du premier octet immédiatement après le dernier élément de la file, la prochaine opération `put` (voir ci-dessous) stockera un nouvel élément à cette adresse.

De cette description il découle que :

1. `occupe == libre` si et seulement si il n'y a aucun élément dans la file,
2. `(char *) libre - (char *) occupe` divisé par `te` donne le nombre d'éléments actuellement dans la file,
3. `(char *) last - (char *) first` divisé par `te` donne la capacité de la file,
4. `libre == last` s'il n'y a plus de place pour un nouvel élément (l'adresse `libre` n'étant plus une adresse dans l'espace de stockage, il faudra réorganiser la file pour pouvoir y insérer un nouveau élément).

Remarques. Noter les conversions en `char *` pour calculer, en nombre d'octets, la distance entre `libre` et `occupe`, ou encore entre `last` et `first`. L'arithmétique des pointeurs est interdite sur le type `void *` en C standard, mais `gcc` la tolère en effectuant implicitement ces conversions préalables sans qu'il soit nécessaire de les écrire explicitement. Nous nous en tiendrons au C standard dans cet énoncé, *c.f.* le tout premier exercice.

La figure ci-dessous représente une file de nombres entiers. La capacité de la file – le nombre maximal d'éléments stockables – est de 7 éléments La file contient trois éléments : $3, -8, 22$. Et La prochaine opération `get` renverra le premier élément 3 qui sera retiré de la file. La prochaine opération `put` ajoutera un nouvel élément après 22.



1. Ecrire les deux fonctions suivantes :

```
static void *decale(void *f, size_t d)
static ptrdiff_t diff(void *f, void *g)
```

La fonction `decale` doit renvoyer l'adresse `f` décalée de `d` octets. La fonction `diff` suppose `f < g`, et doit renvoyer la distance en octets entre `f` et `g`.

Comme indiqué ci-dessus, pour rester sur la partie standard du C, les deux fonctions doivent convertir en `char *` leurs arguments en `void *` pour calculer leur résultats. Dans la suite, servez-vous de ces fonctions pour éviter d'alourdir le code par trop de conversions vers `char *`.

2. Écrire la fonction

```
fifo *create_fifo(size_t capacite, size_t te)
```

Cette fonction doit allouer l'espace de stockage d'une file de capacité `capacite`, et de taille d'élément `te`, allouer sa structure, puis renvoyer l'adresse de la stricture allouée. Initialement, on aura `first == libre == occupe`.

3. Écrire la fonction

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

qui libère tout l'espace mémoire alloué pour une file.

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 `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 s'il y a de la place à la fin de la file, c'est-à-dire si `libre < last`. Le paramètre `pelem` contient l'adresse de l'élément qui sera recopié à l'adresse `libre`. La fonction renverra l'adresse du nouvel 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 renverra `NULL` et le nouvel élément ne sera pas ajouté à la file.

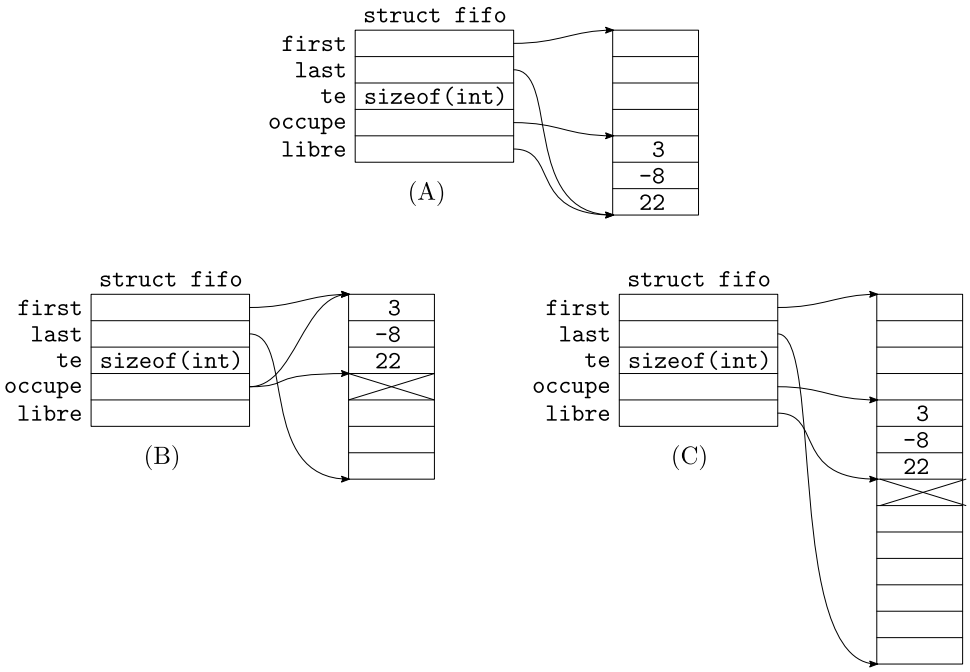
7. Écrire la fonction

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

qui ajoute un nouvel élément à la fin de la file, même si celle-ci est pleine. Si la file n'est pas pleine, servez-vous de la fonction précédente. Sinon, c'est-à-dire si `libre == last` comme dans la figure (A) dessous, il y a deux algorithmes possibles pour ajouter ce nouveau élément.

(B) Le premier algorithme déplace (`memmove`) tous les éléments au début du tableau comme dans la figure (B) ci-dessous. Le nouvel élément sera ajouté à la position marquée. Cet algorithme est applicable uniquement si le nombre courant d'éléments de la file est strictement inférieur à sa capacité. Noter que, une fois le déplacement des éléments et les ajustements de pointeurs effectués, l'insertion peut se faire avec la fonction précédente.

(C) Le deuxième algorithme double la taille de l'espace de stockage (`realloc`), voir la figure (C) ci-dessous. Une fois cet espace agrandi et les pointeurs ajustés, l'ajout peut se faire avec la fonction de la question précédente.



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

```
diff(libre, occupe) < diff(last, first) * 0.25
```

Dans le cas contraire, appliquez le premier algorithme.