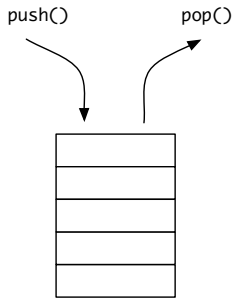
Exercice facultatif à rendre : pile

1 Piles

Une pile est une structure de donnée n'autorisant que deux opérations `push()` et `pop()`. Les données sont empilées (donc en haut de la pile) à l'aide de l'opération `push()`. Elles sont retirées en utilisant l'opération `pop()`.

Il s'agit d'une structure LIFO, *Last In First Out* : le dernier élément ajouté sera le premier enlevé. Ainsi, si les valeurs 33 et 42 sont empilées dans cet ordre, alors elles seront dépilées dans l'ordre inverse : 42 puis 33.



2 À rendre

Vous devez écrire une pile ainsi que les fonctions de manipulation associées. Les signatures des fonctions sont imposées. Le type des éléments stockés dans la pile est `unsigned int`.

Les piles créées **doivent avoir une taille maximale fixée**. Ainsi, si une pile est créée pour stocker 100 éléments, alors elle ne pourra stocker que de 0 à 100 éléments.

Vous trouverez avec cet énoncé le fichier `stack.c`. Celui-ci contient des définitions de structures et fonctions vides, marquées par `TO DO`. Votre travail est de remplir ces zones laissées vides en suivant les instructions mises en commentaires.

Ces fonctions sont rappelées ici :

```
// The stack definition.
// A stack has a fixed maximal size.
typedef struct
{
  // TO DO
} uint_stack_t;
```

```
// Create a new stack of size nb.
// Return NULL if the creation didn't succeed (for a lack of memory).
uint_stack_t* uint_stack_create(size_t nb)
{
```

```
  // TO DO
}
```

```
// Free the memory used by a stack.
// After this call, the free'd stack can no longer be used.
void uint_stack_destroy(uint_stack_t* stack)
{
  // TO DO
}
```

```
// Copy src into dst.
// Return -1 if src is too big to be copied into dst, 0 otherwise.
int uint_stack_copy(uint_stack_t* src, uint_stack_t* dst)
{
  // TO DO
}
```

```
// Swap the content of two stacks.
void uint_stack_swap(uint_stack_t* stack1, uint_stack_t* stack2)
{
  // TO DO
}
```

```
// Push a value.
// If the stack is full, it's not modified.
// Return 0 if the stack wasn't full, -1 otherwise.
int uint_stack_push(uint_stack_t* stack, unsigned int element)
{
  // TO DO
}
```

```
// Pop a value.
// If the stack is empty, it's not modified.
// Return 0 if the stack wasn't empty, -1 otherwise.
int uint_stack_pop(uint_stack_t* stack, unsigned int* element)
{
  // TO DO
}
```

```
// Return the size of a stack.
// This is the actual number of stored elements, not the maximal size.
size_t uint_stack_size(uint_stack_t* stack)
{
  // TO DO
}
```

3 Notation

La notation, sur 20, sera effectuée sur les trois critères suivants :

— Le nombre de tests (voir section 5) qui passe sans erreur
— L'absence de fuites mémoires [1]
— La lisibilité du code

Ce dernier critère, bien que subjectif, est très facile à respecter : indentez votre code, utilisez des noms de variable significatifs, et ne mettez qu'une instruction par ligne.

La note sera ramenée sur 2 points en tant que bonus sur le projet (de manière individuelle).

4 Consignes

Le rendu doit-être sur le LMS (`http://lms.isae.fr/mod/assign/view.php?id=18293`) , au plus tard le **jeudi 26 mars 2015, à 23 h 55**.

Le rendu se fait sous la forme d'un fichier nommé `FOOBAR_stack.c` où `FOOBAR` est votre nom de famille.

Le non-respect des règles suivantes impliquera que votre code **ne sera pas corrigé** :

— Votre code doit compiler avec la ligne de commande suivante, sous l'environnement Linux des salles F116 et F117 : `clang -Wall -Werror -std=c99 -O0 -g FOOBAR_stack.c`.
— Les signatures des fonctions et de la structure ne doivent pas changer.

5 Conseils

— Vous trouverez dans la fonction `main()` du fichier `stack.c` un ensemble de tests vérifiant votre code. Ils vous aideront à bien comprendre le comportement attendu des fonctions. Bien évidemment, tous les tests utilisés pour la notation ne sont pas forcément présents ! Écrivez donc les vôtres.
— Réfléchissez bien à la structure que vous allez utiliser pour définir les piles : rappelez-vous qu'elles ont une **taille maximale fixe**.

1. Déterminées par l'outil valgrind (`http://valgrind.org`).