TP4 : Pile et équations Postfix

1 Définir une bibliothèque de fonctions sur les piles

Nous allons définir une bibliothèque de fonctions permettant de gérer une pile de double.

1. Tout d'abord, nous allons définir une structure de maillon et une structure de pile :

```c
typedef struct maillon
{
    double value;
    struct maillon *next;
} maillon;

typedef struct pile
{
    maillon *tete;
} pile;
```

2. A vous d'écrire les fonctions suivantes permettant de gérer une pile :
   - **pile *Pile_new()** qui initialise une nouvelle pile.
   - **void Pile_free(pile *p)** qui permet de supprimer de la mémoire une pile ainsi que tous les maillons de la pile.
   - **void Pile_push(pile *p, double v)** qui rajoute la valeur v en tête de pile.
   - **double Pile_pop(pile *p)** qui retire le maillon en tête de la pile et renvoie sa valeur.
   - **void Pile_affiche(pile *p)** qui affiche tout le contenu de la pile.
   - **_Bool Pile_isEmpty(pile *p)** qui renvoie 1 si la pile est vide, et 0 sinon.

3. Testez votre pile, en y insérant et retirant des éléments et en affichant son contenu à chaque étape.

4. Compilez votre programme avec l'option -g :

```
gcc -g monprog.c -o monprog
```

   Puis, à l'aide de la commande

```
valgrind --tool=memcheck --leak-check=full monprog
```

   vérifiez que votre programme (qui s'appelle, dans cet exemple, **monprog**) n'a pas de fuite mémoire : toute la mémoire allouée doit être libérée.

5. Pour finir, déplacez toutes les fonctions concernant la pile dans un fichier *mypile.c*, et toutes les déclarations de structures et les include nécessaires dans un fichier *mypile.h*. La fonction **main** effectuant les tests de la pile reste dans le fichier où elle était (par exemple,**prog.c**). Rajoutez, dans le fichier header (le fichier**mypile.h**), tous les entêtes des fonctions du fichier source correspondant. Enfin, rajoutez dans le fichier *mypile.c*, ainsi que dans le fichier principal de votre programme, la ligne

```c
#include "myliste.h"
```

   Pour compiler votre programme c, vous rajouterez le nom du fichier source, contenant les codes gérant la liste, à la liste des fichiers à compiler :

```
gcc -g -Wall myprog.c mypile.c -o prog
```

Testez votre programme afin de vérifiez qu'il a toujours le même comportement. Grâce à ce système, vous n'êtes plus obligé de placer tout votre code dans le même fichier, vous pouvez créer différents modules de codes.

2 Résoudre une équation postfix (notation polonaise)

La notation postfix, parfois appelée polonaise inversée, consiste à placer d'abord les opérandes puis les opérateurs dans une équation, et éviter ainsi d'utiliser des règles de priorité ou des parenthèses.

Par exemple, pour multplier 3 et 2, on note $32*$.

Si on souhaite réaliser l'opération $3*(2+5)$, on écrirait $25+3*$, ce qui se lit comme "prendre 2 et 5, les additionner, prendre 3, et le multiplier avec le résultat précédent".

L'équation $3*(2+5)+9*2$ consiste d'abord à réaliser l'addition de 2 et de 5, puis à multiplier le résultat par 3, puis à multiplier 9 et 2, et l'additionner au résultat précédent. En notation postfix, on écrit ces opérations de gauche à droite : $25+3*92*+$.

Nous allons écrire un programme qui, à l'aide d'une pile, réalise ces opérations.

1. Considérez le code suivant :

```c
int main(int argc, char* argv[])
{
    if (argc !=2)
    {
        printf("usage: %s <equation>\n", argv[0]);
        return EXIT_FAILURE;
    }

    printf("%s\n", argv[1]);
    return EXIT_SUCCES;
}
```

   Ce morceau de code permet de tester le nombre de paramètre passés au programme principal : si l'utilisateur n'a pas, lors de l'appel du programme, spécifié un paramètre supplémentaire, un message l'invitant à saisir une équation en paramètre de la fonction est affiché.
   Sinon, c'est l'équation saisie qui est affichée : le programme devra afficher le résultat du calcul demandé. Par exemple, si l'utilisateur appelle le programme ainsi

```
monprog 23*4+
```

   On s'attend à ce que le programme affiche 10 à l'écran. L'équation passée en paramètre du programme ne contiendra que l'une de quatre opérations de base, ainsi que des chiffres de 0 à 9 (nous ne traiterons pas les nombres avec plus d'un chiffre dans ce programme).
   Testez le morceau de code précédent...

2. Pour résoudre l'équation, on suivra cette démarche :

```
Initialiser une pile p vide;
pour tous les caractères c dans l'équation à résoudre faire
    si c est un chiffre alors
        Convertir c en valeur numérique et l'empiler dans p;
    fin
    sinon si c est une opération alors
        Dépiler les deux dernières valeurs stockées dans p, et réaliser l'opération entre elles ;
        Empiler le résultat dans p ;
    fin
fin
A la fin, la pile ne doit contenir qu'une seul valeur et c'est le résultat de l'équation.
```

   Réalisez ce programme en C et testez le avec différentes équations pour vérifier que vous obtenez le bon résultat.

3. Rajoutez un opérateur ĝui réalise la puissance.

4. Rajoutez un opérateur v qui réalise la racine carrée... Attention, de combien d'opérandes cet opérateur a besoin ?

5. Comment, dans votre code, avertir l'utilisateur s'il a écrit une équation incorrecte (par exemple, 45*+) ?

6. Pourquoi travailler avec des nombres de plus d'un chiffre pose problème ? Comment le résoudre (on vous demande simplement d'y réfléchir, pas d'implémenter la solution) ?

3 Terminez l'exercice sur les listes du TP précédent