TD N° 3 « Les Piles & Files»

Déclaration d'une Pile de caractères :

```
typedef struct elt {
        char ch ;
        struct elt *suiv;
} Node ;
int estVide(Node* p) ;
void empiler(Node *p, char c) ;
char depiler(Node *p) ;
char consulter(Node *p) ;
```

Exercice N° 1 : Implémentation de la pile

Donner le code des fonctions « estVide», « empiler », « depiler » et « consulter ».

Correction :

```c
#include<stdio.h>
#include<stdlib.h>

typedef struct elt{
    char ch;
    struct elt* suiv;
} Node;

int estVide(Node *p){
    return p==NULL;
}

Node* empiler(Node *p, char c){

    // Créer le noeud
    Node *nv = malloc(sizeof(Node));
    if(nv==NULL) {
        printf("Erreur d'allocation de mémoire");
        exit(1);
    }
    nv->ch = c;
    nv->suiv = p;
    return nv;
}
```

```c
char consulter(Node *p){
    if(estVide(p)){
        printf("La liste est vide!");
        exit(1);
    }
    return p->ch;
}

Node* depiler(Node *p, char *r){
    if(estVide(p)){
        printf("Liste vide!");
        return NULL;
    }
    *r = p->ch;
    Node *tmp = p;
    p = p->suiv;
    free(tmp);
    return p;
}

char depilerV2(Node **p){

    if(estVide(*p)){
        printf("Liste vide!");
        exit(1);
    }
    Node *tmp = *p;
    char r = tmp->ch;
    *p = tmp->suiv;
    free(tmp);

    return r;
}
```

Exercice N° 2 : Analyse Syntaxique

Il s'agit de vérifier si une expression est bien parenthésé.

Nous devons écrire un programme qui :

- Accepte les mots comme : (xxx),(xxx)(xx(xxx)x) ou (xx(xx(xxx)(xxx)xx)(xxx)xx)
- Rejette les mots comme : xx)xx(xx,(xxx)xxx)xx(xx(xxx)xx ou (((xxx)x(xxx))))

NB : Utilisez la pile du premier exercice.

Correction :

```c
int main(){
```

```c
    // Déclaration de la pile
    Node *p = NULL;

    // Lire la phrase parenthésée
    char s[100];
    printf("Donner une chaine de caractères :");
    scanf("%s", s);

    // Parcourir la chaîne de caractères
    for(int i=0; s[i]!='\0'; i++){
        if(s[i]=='('){
            p = empiler(p, s[i]);
        }
        else if(s[i]==')'){
            if(estVide(p)){
                printf("Parenthèses mal appariées!\n");
                exit(1);
            }
            char r;
            p = depiler(p, &r);  // ou r = depilerV2(&p);
    }

    if(!estVide(p)){
        printf("Parenthèses mal appariées!\n");
        return 1;
    } else {
        printf("Parenthèses bien appariées!\n");
    }

    return 0;
}
```

Exercice N° 3 : Expressions arithmétiques

Une application courante des piles se fait dans le calcul arithmétique :

L'ordre dans la pile permet d'éviter l'usage des parenthèses. La notation postfixée (polonaise) consiste à placer les opérandes devant l'opérateur. La notation infixée (parenthésée) consiste à entourer les opérateurs par leurs opérandes. Les parenthèses sont nécessaires uniquement en notation infixée.

La notation postfixée est d'un emploi plus facile puisqu'on sait immédiatement combien d'opérandes il faut rechercher.
Détaillons ici la saisie et l'évaluation d'une expression postfixée :
La notation usuelle, comme (3 + 5) * 2, est dite infixée. Son défaut est de nécessiter l'utilisation de parenthèses pour éviter toute ambiguïté (ici, avec 3 + (5 * 2)). Pour éviter le parenthésage, il est possible de transformer une expression infixée en une expression postfixée en faisant "glisser" les opérateurs arithmétiques à la suite des expressions auxquelles ils s'appliquent.

Exemples :

- (3 + 5) * 2 s'écrira en notation postfixée (notation polonaise) : 3 5 + 2 *
  Alors que 3 + (5 * 2) s'écrira: 3 5 2 * +
- Notation infixe: (A * B)/C En notation postfixe est: AB * C/.
- Notation postfixe : 6 5 2 3 + 8 * + 3 + * En notation infixe est : (6 * ((5 + ((2 + 3) * 8)) + 3))

Travail à faire :

Ecrire un programme lit une expression arithmétique postfixée et affiche le résultat du calcule.

Pour cela nous allons créer pile d'entier avec un tableau d'entiers, et une structure qui modélise la pile :

```
#define MAX 100  // taille maximale de la pile
typedef struct {
    int elements[MAX];
    int sommet;  // indice du dernier élément
} Pile;
int estVide(Pile *p) ;
int estPleinne(Pile *p) ;
void empiler(Pile *p, int n) ;
int depiler(Pile *p) ;
int consulter(Pile *p) ;
```

Correction :

```
#define MAX 100  // taille maximale de la pile
typedef struct {
      int elements[MAX];
      int sommet;  // indice du dernier élément
} Pile;

int estVide(Pile *p){
    return p->sommet == -1;
}
int estPleinne(Pile *p) {
    return p->sommet == MAX - 1;
}
void empiler(Pile *p, int n){
    if (estPleinne(p)) {
        printf("Pile pleine");
        exit(1);
    }
    p->sommet++;
    p->elements[p->sommet] = n;
}

int depiler(Pile* p) {
```

```
    if (estVide(p)) {
        printf("Pile vide");
        exit(1);
    }
    int val = p->elements[p->sommet];
    p->sommet--;
    return val;
}


// programme principal pour evaluer expression postfixee
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {

    char expr[] = "23*54*+9-";  // expression postfixée à évaluer

    Pile p;
    p.sommet = -1;  // initialisation de la pile

    for (int i = 0; i < strlen(expr); i++) {
        char c = expr[i];

        if (c>='0' && c<='9') {
            empiler(&p, c - '0');  // convertir le caractère en entier
        } else {
            int a = depiler(&p);
            int b = depiler(&p);
            int result;

            switch (c) {
                case '+': result = b + a; break;
                case '-': result = b - a; break;
                case '*': result = b * a; break;
                case '/':
                    if (a == 0) {
                        printf("Erreur: division par zéro\n");
                        exit(1);
                    }
                    result = b / a;
                    break;
                default:
                    printf("Opérateur inconnu: %c\n", c);
                    exit(1);
            }
            empiler(&p, result);
        }
```

```
    }
    int resultat = depiler(&p);  // le résultat final
    printf("Le résultat de l'expression postfixée %s est: %d\n", expr, resultat);
    return 0;
}
```

Exercice N° 4 : Une file d'attente des clients

Nous allons écrire un petit programme qui simule **une file d'attente dans une banque**.

On veut gérer une file d'attente de clients à la banque.
Chaque client est représenté par son nom.
Quand un client arrive, il entre dans la file.
Quand un guichet est libre, on sert le client en tête de file (le premier arrivé).

Nous allons utiliser la structure suivante :

```
#define MAX 100  // taille maximale de la file
typedef struct {
   char elements[MAX][30];
   int tete;   // indice du premier élément
   int queue;  // indice du dernier élément
   int taille; // nombre d'éléments actuels
} File;

void initialiserFile(File *f) ;
int estVide(File *f) ;
int estPleine(File *f) ;
void enfiler(File *f, char *nom) ;
char* defiler(File *f) ;
char* consulter(File *f) ;
void afficherFile(File *f)
```

Tâches à faire :

1. Implémenter les primitives de la file.
2. Dans une fonction main :
    - Ajouter 5 clients dans la file.
    - Afficher la file complète.
    - Servir (retirer) 2 clients et afficher la file après chaque retrait.
    - Afficher le client qui est maintenant en tête de la file (sans le retirer).

Correction :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```c
#define MAX 100  // taille maximale de la file
typedef struct {
    char elements[MAX][30];
    int tete;   // indice du premier élément
    int queue;  // indice du dernier élément
    int taille; // nombre d'éléments actuels
} File;

void initialiserFile(File *f) {
    f->tete = 0;
    f->queue = -1;
    f->taille = 0;
}

int estVide(File *f) {
    return f->taille == 0;
}
int estPleine(File *f) {
    return f->taille == MAX;
}
void enfiler(File *f, char *nom){
    if (estPleine(f)) {
        printf("La file est pleine\n");
    } else {
        f->queue = (f->queue + 1) % MAX;
        strcpy(f->elements[f->queue], nom);
        f->taille++;
    }
}

char* defiler(File *f) {
    if (estVide(f)) {
        printf("La file est vide\n");
        return NULL;
    } else {
        char *nom = f->elements[f->tete];
        f->tete = (f->tete + 1) % MAX;
        f->taille--;
        return nom;
    }
}

char* consulter(File *f) {
    if (estVide(f)) {
        printf("La file est vide\n");
        return NULL;
    } else {
        return f->elements[f->tete];
    }
```

```c
}

void afficherFile(File *f) {
    if (estVide(f)) {
        printf("La file est vide\n");
    } else {
        printf("Contenu de la file:\n");
        for (int i = 0; i < f->taille; i++) {
            int index = (f->tete + i) % MAX;
            printf("%s\n", f->elements[index]);
        }
    }
}

int main(){
    File f;
    initialiserFile(&f);

    // Ajouter 5 clients dans la file
    enfiler(&f, "Client1");
    enfiler(&f, "Client2");
    enfiler(&f, "Client3");
    enfiler(&f, "Client4");
    enfiler(&f, "Client5");

    // Afficher la file complète
    printf("File complète:\n");
    afficherFile(&f);

    // Servir (retirer) 2 clients et afficher la file après chaque retrait
    for (int i = 0; i < 2; i++) {
        char *servi = defiler(&f);
        if (servi != NULL) {
            printf("Client servi: %s\n", servi);
        }
        printf("File après retrait %d:\n", i + 1);
        afficherFile(&f);
    }

    // Afficher le client qui est maintenant en tête de la file (sans le retirer)
    char *teteClient = consulter(&f);
    if (teteClient != NULL) {
        printf("Client en tête de la file: %s\n", teteClient);
    }

    return 0;
}
```