```c
/**
 * Ce programme prend dans ses paramètres programme une expression
 * en notation postfixée et affiche le résultat de son évaluation sur
 * la sortie standard; Les opérandes sont des nombres et les opérateurs
 * les quatre opérations + - * /.
 *
 * @author V. Granet (vg@unice.fr)
 */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdbool.h>
#include <assert.h>
#include "pile.h"

/*
 * Rôle : renvoie 1 si la chaîne s est une suite de chiffres (base 10)
 *        et 0 sinon
 */
bool estUnEntier(const char *s) {
  if (*s=='\0') return false;
  if (*s=='+'||*s=='-') s++;
  // *s doit être un chiffre, le premier de l'entier
  do {
    if (!isdigit(*s)) return false;
  } while (*++s!='\0');
  return true;
}

/*
 * Rôle : renvoie 1 la chaîne op représente le caractère '+', '-', 'x'
 *        ou '/' et 0 sinon
 */
bool estUnOpérateur(const char *s) {
  return (*s=='+'||*s=='-'||*s=='x'||*s=='/') && *(s+1)=='\0';
}

int getInt(const void *pi) {
  return *((int *) pi);
}

int *makeInt(const int n) {
  int *p = malloc(sizeof(int));
  *p = n;
  return p;
}

int main(int argc, char *argv[])
{
  if (argc==1) return EXIT_SUCCESS;
  // else évaluer l'expression postfixée
  pile p = pileVide();
  for (int i=1; i<argc; i++) {
    if (estUnEntier(argv[i]))
      // traiter l'opérande entier
      empiler(&p, makeInt(atoi(argv[i])));
    else
      if (estUnOpérateur(argv[i])) {
        int opG, opD;
        // prendre dans la pile les opérandes
        //  droit et gauche de l'opérateur
        opD = getInt(sommet(p)); dépiler(&p);
        opG = getInt(sommet(p)); dépiler(&p);
        // évaluer l'opérateur et empiler
        // le résultat
        switch (*argv[i]) {
```



```c
        case '+' : empiler(&p, makeInt(opG+opD)); break;
        case '-' : empiler(&p, makeInt(opG-opD)); break;
        case 'x' : empiler(&p, makeInt(opG*opD)); break;
        case '/' :
          assert(opD!=0);
          empiler(&p, makeInt(opG/opD)); break;
        }
      }
      else {
        fprintf(stderr, "%s élément inconnu\n", argv[i]);
        return EXIT_FAILURE;
      }
  } // fin des paramètres programme
  // la pile ne contient qu'un seul élément => résultat
  // => l'afficher
  int res = getInt(sommet(p));
  dépiler(&p);
  if (estVide(p)) {
    // ok
    printf("=> %d\n", res);
    return EXIT_SUCCESS;
  }
  else {
    fprintf(stderr, "Erreur : expression mal formée\n");
    return EXIT_FAILURE;
  }
}
```

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

#include "liste.h"

/*
 * Rôle : renvoie une liste vide
 */
liste listeVide(void) {
  return NULL;
}

/*
 * Rôle : renvoie la longueur de la
 *        liste l
 */
int longueur(const liste l) {
  pnoeud p = l;
  int lg = 0;
  while (p!=NULL) {
    lg++;
    p = p->suivant;
  }
  return lg;
}

/*
 * Antécédent : 1<=r<=longueur(l)
 * Rôle : renvoie l'élément au rang r
 */
T ième(const liste l, const int r) {
  assert(r>=1 && r<=longueur(l));
  pnoeud p = l;
  for (int i=1; i<r; i++)
    p = p->suivant;
  // p désigne le noeud de rang r
  return p->elt;
}

/*
 * Rôle : crée un noeud avec l'élément e
 *       et renvoie un pointeur sur ce noeud
*/
static struct noeud* créerNoeud(const T e) {
  struct noeud *n = malloc(sizeof(struct noeud));
  n->elt = e;
  n->suivant = NULL;
  return n;
}

/*
 * Rôle : insère dans la liste l l'élément e, au rang r
 * Antécédent : 1 $\leqslant$ r $\leqslant$ longueur(l) + 1
 */
void insérer(liste *l, const int r, const T e) {
  assert(r>=1 && r<=longueur(*l)+1);
  // le rang r est valide
  struct noeud *n = créerNoeud(e);
  if (r==1) { // insertion en tête de liste
    n->suivant = *l;
    *l = n;
  }
  else { // cas général, r>=2
    struct noeud *q = *l;
    for (int i=2; i<r; i++)
```



```c
      q = q->suivant;
    // q désigne l'élément de rang r-1
    n->suivant = q->suivant;
    q->suivant = n;
  }
}

/*
 * Rôle : supprime l'élément de rang r de la liste l
 * Antécédent : 1 $\leqslant$ r $\leqslant$ longueur(l)
 */
void supprimer(liste *l, const int r) {
  assert(r>=1 && r<=longueur(*l));
  // le rang r est valide
  struct noeud *p;
  if (r == 1) {
    // suppression en tête de liste
    p = *l;
    *l = (*l)->suivant;
  }
  else { // cas général, r>=2
    struct noeud *q = *l;
    for (int i=2;  i<r; i++)
      q = q->suivant;
    // q désigne l'élément de rang r-1
    p = q->suivant;
    q->suivant = p->suivant;
  }
  free(p);
}
```

```c
#pragma once

typedef void * T;

typedef struct noeud {
  T elt ;
  struct noeud *suivant;
} * pnoeud;

typedef pnoeud liste;

extern liste listeVide(void);
extern int longueur(const liste);
extern T ième(const liste, const int);
extern void insérer(liste *, const int, const T);
extern void supprimer(liste *, const int);
```



```
CC = gcc              # le compilateur à utiliser
CFLAGS = -Wall # les options du compilateur
LDFLAGS =           # les options pour l'éditeur de liens
SRC = calc.c pile.c liste.c # les fichiers sources
PROG = calc          # nom de l'exécutable
OBJS =  $(SRC:.c=.o) # les .o qui en découlent
.SUFFIXES: .c .o     # lien entre les suffixes

all: $(PROG)

# étapes de compilation et d'édition de liens
# $@ la cible  $^ toutes les dépendances
$(PROG): $(OBJS)
        $(CC)  -o $@ $^ $(LDFLAGS)

# le lien entre .o et .c
# $< dernière dépendance
%.o: %.c
        $(CC) $(CFLAGS) -c $<

# pour faire propre
.PHONY: clean
clean:
        rm -f *.o *~ core a.out $(PROG)
```

```c
#include <stdbool.h>
#include <assert.h>
#include "pile.h"

pile pileVide(void) {
  return listeVide();
}

bool estVide(const pile p) {
  return longueur(p)==0;
}

T sommet(const pile p) {
  assert(!estVide(p));
  return ième(p, 1);
}

void empiler(pile *p, const T e) {
  insérer(p, 1, e);
}

void dépiler(pile *p) {
  assert(!estVide(*p));
  supprimer(p, 1);
}
```



```c
#pragma once

#include <stdbool.h>
#include "liste.h"

typedef liste pile;

extern pile pileVide(void);
extern bool estVide(const pile);
extern T sommet(const pile);
extern void empiler(pile *, const T);
extern void dépiler(pile *);
```