Prog. Or. Système - Correction série 09 : Pointeurs (5)

Exercice 1 : Explorer la mémoire (revisité)
(fichier src/memory_view_2.c)

```
// C99
#include <stdio.h>

typedef unsigned char octet;

// ===========================================================================
static inline void affiche_bit(const octet c,
                               const octet position_pattern)
{
  putchar(c & position_pattern ? '1' : '0');
}

// ===========================================================================
void affiche_binaire(const octet c) {
  for(octet mask = 0x80; mask; mask >>= 1)
    affiche_bit(c, mask);
}

// ===========================================================================
void dump_mem(const octet* ptr, size_t length)
{
  const octet* const end = ptr + length;
  for (const octet* p = ptr; p < end; ++p) {
    printf("%p : ", p);
    affiche_binaire(*p);
    printf(" %3u ", (unsigned int) *p);
    if ((*p >= 32) && (*p <= 126)) {
      printf("'%c'", *p);
    }
    putchar('\n');
  }
  puts("----------------------------------------");
}

// ===========================================================================
int main(void)
{
  int a = 64 + 16;
  int b = -a;
  double x = 0.5;
```

```
  double y = 0.1;

  dump_mem( (octet*) &a, sizeof(a) );
  dump_mem( (octet*) &b, sizeof(b) );
  dump_mem( (octet*) &x, sizeof(x) );
  dump_mem( (octet*) &y, sizeof(y) );

  return 0;
}
```

Exercice 2 : piles et parenthèses

Suivre la démarche indiquée dans l'exercice.

Voici un résultat possible :

(fichier src/piles.c)

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

typedef char type_el;

/* version simpliste avec tableau de taille fixe.
   Une version plus réaliste utiliserait des tableaux dynamique.
*/

#define STACK_OVERFLOW 256
#define VIDE -1
typedef struct {
  type_el tab[STACK_OVERFLOW];
  int tete;
} Pile;

/* Prototypes des fonctions */
int empile(Pile* p, type_el e);
void depile(Pile* p);
type_el top(Pile* p);
int est_vide(Pile* p);
void init_Pile(Pile* p);

/* vérification de parenthèsage */
int check(char* parentheses);

/* ---------------------------------------------------------------- */
#define MAX 1024
int main(void)
{
```

```
  char s[MAX+1];
  int taille_lue;

  do {
    printf("Entrez une expresssion parenthèsée : ");
    fgets(s, MAX, stdin);
    taille_lue = strlen(s) - 1;
    if ((taille_lue >= 0) && (s[taille_lue] == '\n'))
      s[taille_lue] = '\0';
    if (s[0] != '\0')  /* pas vide */
      printf(" -> %s\n", check(s) ? "OK" : "Erreur");
  } while ((taille_lue < 1) && !feof(stdin));

  return 0;
}

/* ------------------------------------------------------------------- */
int empile (Pile* p, type_el e)
{
  ++(p->tete);
  if (p->tete >= STACK_OVERFLOW) {
    p->tete = STACK_OVERFLOW;
    return 0; }
  else {
    p->tab[p->tete] = e;
  }
  return 1;
}

/* ------------------------------------------------------------------- */
void depile (Pile* p)
{
  if (!est_vide(p)) --(p->tete);
}

/* ------------------------------------------------------------------- */
type_el top (Pile* p)
{
  if (!est_vide(p))
    return p->tab[p->tete];

  else /* que faire ??  -> totalement arbitraire... */
    return 0;
}

/* ------------------------------------------------------------------- */
int est_vide(Pile* p)
```

```
{
  return ((p->tete < 0) || (p->tete >= STACK_OVERFLOW));
}

/* ------------------------------------------------------------------------ */
void init_Pile(Pile* p)
{
  p->tete = -1; /* convention => pile vide */
}

/* ------------------------------------------------------------------------ */
int check(char* s) {
  Pile p;
  unsigned int i;

  init_Pile(&p);

  for (i = 0; i < strlen(s); ++i) {
    if ((s[i] == '(') || (s[i] == '['))
      empile(&p, s[i]);
    else if (s[i] == ')') {
      if ((!est_vide(&p)) && (top(&p) == '('))
        depile(&p);
      else
        return 0;
    } else if (s[i] == ']') {
      if ((!est_vide(&p)) && (top(&p) == '['))
        depile(&p);
      else
        return 0;
    }
  }

  return est_vide(&p);
}
```

Il est vrai que les fonctions `empile`, `depile`, `top` et `est_vide` sont relativement triviales avec cette implémentation(/réalisation) des piles.
Tellement trivial que certains d'entre vous ont peut être directement utilisé leur définition dans le code de la fonction `check`.

Mais je préfère que vous prenniez l'habitude d'avoir cette approche d'analyse descente : décomposer le problèmes en problèmes plus simples et recommencer. Donc ici se dire : "j'ai besoin de piles. C'est quoi une pile ? ... -> 4 fonctions" et donc on écrit ces fonctions.

Cela a l'avantage d'être plus facilement maintenable (imaginez par exemple que demain vous reprenniez ce code mais que vous décidiez de changer l'implémentation des piles. Vous n'aurez alors qu'à changer les 4 fonctions ci-dessus et non pas l'équivalent de chacun de leur appel partout dans le code).

Exercice 3 : piles et polonaise
Voici un exemple de solution, utilisant les même piles que ci-dessus.
Plusieurs améliorations sont possibles.

(fichier src/piles2.c)

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

typedef int type_el;

/* version simpliste avec tableau de taille fixe.
   Une version plus réaliste utiliserait des tableaux dynamique.
*/

#define STACK_OVERFLOW 256
#define VIDE -1
typedef struct {
  type_el tab[STACK_OVERFLOW];
  int tete;
} Pile;

/* Prototypes des fonctions */
int empile(Pile* p, type_el e);
void depile(Pile* p);
type_el top(Pile* p);
int est_vide(Pile* p);
void init_Pile(Pile* p);

int eval(char* entree);

/* ------------------------------------------------------------------- */
int empile (Pile* p, type_el e)
{
  ++(p->tete);
  if (p->tete >= STACK_OVERFLOW) {
    p->tete = STACK_OVERFLOW;
    return 0; }
  else {
    p->tab[p->tete] = e;
  }
  return 1;
}
```

```
/* ------------------------------------------------------------------- */
void depile (Pile* p)
{
  if (!est_vide(p)) --(p->tete);
}

/* ------------------------------------------------------------------- */
type_el top (Pile* p)
{
  if (!est_vide(p))
    return p->tab[p->tete];

  else /* que faire ??  -> totalement arbitraire... */
    return 0;
}

/* ------------------------------------------------------------------- */
int est_vide(Pile* p)
{
  return ((p->tete < 0) || (p->tete >= STACK_OVERFLOW));
}

/* ------------------------------------------------------------------- */
void init_Pile(Pile* p)
{
  p->tete = -1; /* convention => pile vide */
}

/* ------------------------------------------------------------------- */
#define MAX 1024
int main(void)
{
  char s[MAX+1];
  int taille_lue;

  do {
    printf("Entrez une expresssion à évaluer : ");
    fgets(s, MAX, stdin);
    taille_lue = strlen(s) - 1;
    if ((taille_lue >= 0) && (s[taille_lue] == '\n'))
      s[taille_lue] = '\0';
    if (s[0] != '\0')  /* pas vide */
      printf(" -> résultat : %d\n", eval(s));
  } while ((taille_lue < 1) && !feof(stdin));

  return 0;
```

```
}

/* ---------------------------------------------------------------------- */
int eval(char* s)
{
  Pile p;
  unsigned int i;

  init_Pile(&p);

  /* recopie dans la pile */
    for (i = 0; i < strlen(s); ++i)
    if ((s[i] >= '0') && (s[i] <= '9')) {
      /* On a lu un chiffre */
      empile(&p, (int) (s[i] - '0'));
    } else if ((s[i] == '+') || (s[i] == '-') ||
               (s[i] == '*') || (s[i] == '/')) {

      int x, y; /* arguments */

      /* recupere le second argument */
      y = top(&p);
      depile(&p);

      /* recupere le premier argument */
      x = top(&p);
      depile(&p);

      /* calcule et empile le resultat */
      switch(s[i]) {
      case '+': empile(&p, x + y); break;
      case '-': empile(&p, x - y); break;
      case '*': empile(&p, x * y); break;
      case '/': empile(&p, x / y); break;
      }
    }

  return top(&p);
}
```

Exercice 4 : Needleman-Wunsch (a.k.a. Viterbi)
(fichier src/needleman-wunsch.c)

```
// C99
```

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

#include <stdint.h> /* pour SIZE_MAX */
#ifndef SIZE_MAX
#define SIZE_MAX (~(size_t)0)
#endif

/* On représente les prédecesseurs par le déplacement relatif */
typedef enum {
        DirDiag, /* en diagonale */
        DirHorz, /* horizontalement */
        DirVert  /* verticalement */
} Dir;

/* Une cellule du tableau */
typedef struct {
        int val; /* la "vraie" valeur à stocker  */
        Dir dir; /* la direction du prédécesseur */
} Cell;

/* Table rectangulaire des valeurs */
typedef struct {
        size_t width;  /* largeur */
        size_t height; /* hauteur */
        Cell*  tab;    /* tableau */
} Table;

/* Solution : ici une solution est un tableau dynamique de transformations.
 * Dans le cas des pointeurs ce pourrait être une liste chaînée par exemple */
typedef struct {
        size_t size;  /* la taille du tableau      */
        Dir* dirs;    /* la suite des déplacements */
} Solution;


Table* computeTable(const char* s1, const char* s2);
Solution* extractSolution(const Table* tab);
void showSolution(Solution* sol, const char* s1, const char* s2);
void freeTable(Table* tab);

Table* computeTable(const char* s1, const char* s2) {
        size_t i,j;

        Table* res = malloc(sizeof(Table));
        if (res==NULL) return NULL;
```

```
        res->width  = strlen(s1) + 1; /* s1 horizontalement */
        res->height = strlen(s2) + 1; /* s2 verticalement */
    if (SIZE_MAX / res->height < res->width) { free(res); return NULL; }
        res->tab = calloc(res->height * res->width, sizeof(Cell));
        if (res->tab==NULL) { free(res); return NULL; }

        for (i = 0; i < res->height; ++i) {
        Cell* c = res->tab + i*res->width; // j = 0
        c->val = 0;
        c->dir = DirVert;
    }
        for (j = 0; j <res->width; ++j) {
        Cell* c = res->tab + j; // i = 0
        c->val = 0;
        c->dir = DirHorz;
    }
        for (i = 1; i < res->height; ++i) {
                for (j = 1; j < res->width; ++j) {
            Cell* c = res->tab + i*res->width + j;

                        int s = (s1[j-1] == s2[i-1]) ? 2 : -1;
                        int diag = res->tab[ (i-1)*res->width + j-1 ].val + s ;
            int horz = res->tab[ (i  )*res->width + j-1 ].val - 2 ;
            int vert = res->tab[ (i-1)*res->width + j   ].val - 2 ;

                        if (diag>horz && diag>vert) {
                                c->val = diag;
                                c->dir = DirDiag;
                        } else if (horz>vert) {
                                c->val = horz;
                                c->dir = DirHorz;
                        } else {
                                c->val = vert;
                                c->dir = DirVert;
                        }
                }
        }
        return res;
}

Solution* extractSolution(const Table* tab) {
        size_t i,j;

        Solution *sol = malloc(sizeof(Solution));
        if (sol==NULL) return NULL;
```

```
        sol->size = 0;
        sol->dirs = calloc(tab->width + tab->height, sizeof(Dir));
        if (sol->dirs==NULL) return NULL;

        i = tab->height ? (tab->height - 1) : 0;
        j = tab->width  ? (tab->width  - 1) : 0;

        while(i>0 || j>0) {
                Dir dir = tab->tab[i*tab->width+j].dir;
                sol->dirs[sol->size]=dir;
                sol->size++;

                switch (dir) {
                        case DirHorz: --j;      break;
                        case DirVert: --i;      break;
                        case DirDiag: --i; --j; break;
                }
        }

        /* Inversion pour remettre les mouvements dans le bon ordre */
        for (i=0; i<sol->size/2; ++i) {
                Dir tmp = sol->dirs[i];
                sol->dirs[i] = sol->dirs[sol->size-1-i];
                sol->dirs[sol->size-1-i] = tmp;
        }

        /* Réduction de la solution à la bonne taille */
        sol->dirs = realloc(sol->dirs, sol->size*sizeof(Dir));

        return sol;
}

void showSolution(Solution* sol, const char* s1, const char* s2) {
        size_t i,j;
        for (i=0,j=0; i<sol->size; ++i) {
                switch (sol->dirs[i]) {
                        case DirDiag: /*continued*/
                        case DirHorz: printf("%c",s1[j]); ++j; break;
                        case DirVert: printf("_");
                }
        }
        printf("\n");
        for (i=0,j=0; i<sol->size; ++i) {
                switch (sol->dirs[i]) {
                        case DirDiag: /*continued*/
                        case DirVert: printf("%c",s2[j]); ++j; break;
                        case DirHorz: printf("_");
```

```
                }
        }
        printf("\n");
}

void freeTable(Table* tab) {
        free(tab->tab);
        free(tab);
}

int main(void) {
        char s1[] = "Bonjour monsieur, quelle heure est-il à votre montre ?";
        char s2[] = "Bonne journée madame, que l'heureuse fillette vous montre le
chemin";

        Table* tab = computeTable(s1,s2);
        Solution* sol = extractSolution(tab);

        showSolution(sol, s1, s2);

        freeTable(tab);
        free(sol->dirs);
        free(sol);

        return 0;
}
```

Dernière mise à jour : Dernière mise à jour le 15 mars 2016
Last modified: Tue Mar 15, 2016