Prog. Or. Système - Correction série 08 : Pointeurs (4)

Exercice 1 : QCM
(fichier src/qcm.c)

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

#define MAX_REP 10

/* Types */
typedef struct {
  char* question;
  char* reponses[MAX_REP]; /* tableau de 10 pointeurs de caractères */
  unsigned int nb_rep;
  unsigned int solution;
}  QCM;

typedef QCM* Examen;

/* Prototypes */
void affiche(QCM const * question);
int demander_nombre(int min, int max);
unsigned int poser_question(QCM const * question);
unsigned int creer_examen(Examen*);
void detruire_examen(Examen*);

/* ================================================================== */
int main(void)
{
  unsigned int note = 0;
  Examen exam = NULL;
  unsigned int taille_examen = creer_examen(&exam);
  unsigned int i;

  for (i = 0; i < taille_examen; ++i)
    if (poser_question(&(exam[i])) == exam[i].solution)
      ++note;

  /* petite astuce pour accorder 'bonne reponse' si
   * l'utilisateur a plusieurs réponses correctes.
   */
  printf("Vous avez trouvé %d bonne", note);
  if (note > 1) putchar('s');
```

```
  printf(" réponse");
  if (note > 1) putchar('s');
  printf(" sur %d.\n", taille_examen);

  detruire_examen(&exam);
  return 0;
}


/* ========================================================================== */
void affiche(QCM const * q)
{
  unsigned int i;
  printf("%s ?\n", q->question);
  for (i = 0; i < q->nb_rep; ++i) {
   /* on affiche i+1 pour éviter de commencer l'énumération des réponses avec 0 */
    printf("    %d- %s\n", i+1, q->reponses[i]);
  }
}

/* ========================================================================== */
int demander_nombre(int a, int b)
{
   int res;

   if (a > b) { res=b; b=a; a=res; }

   do {
       printf("Entrez un nombre entier compris entre %d et %d : ",
              a, b);
       scanf("%d", &res);
    } while ((res < a) || (res > b));
    return res;
}

/* ========================================================================== */
unsigned int poser_question(QCM const * q)
{
  affiche(q);
  /* on transforme le type int retourné par demander_nombre en un unsigned int */
  return (unsigned int) demander_nombre(1, q->nb_rep);
}

/* ========================================================================== */
unsigned int creer_examen(Examen* retour)
{
  unsigned int i;
```

```
/* Pour cet examen, on a 3 QCM, donc il faut allouer l'équivalent de
 * 3 fois la taille d'un QCM dans la mémoire.
 */
*retour = calloc(3, sizeof(QCM));


/* QUESTION 1 */
/* On alloue une taille de 50 caractères pour la question.
 * Note: malloc(50) ou malloc(50*sizeof(char)) revient
 * au même car sizeof(char) est toujours égal à 1.
 */
(*retour)[0].question = malloc(50);
strcpy((*retour)[0].question,
       "Combien de dents possède un éléphant adulte");

(*retour)[0].nb_rep = 5;

for (i = 0; i < (*retour)[0].nb_rep; ++i) {
/* On alloue 10 caractères pour chaque réponse. */
  (*retour)[0].reponses[i] = malloc(10);
}
strcpy((*retour)[0].reponses[0], "32");
strcpy((*retour)[0].reponses[1], "de 6 à 10");
strcpy((*retour)[0].reponses[2], "beaucoup");
strcpy((*retour)[0].reponses[3], "24");
strcpy((*retour)[0].reponses[4], "2");

(*retour)[0].solution = 2;

/* QUESTION 2 */
/* On alloue 80 caractères pour la question. */
(*retour)[1].question = malloc(80);
strcpy((*retour)[1].question,
       "Laquelle des instructions suivantes est un prototype de fonction");

(*retour)[1].nb_rep = 4;

for (i = 0; i < (*retour)[1].nb_rep; ++i) {
  /* On alloue 14 caractères pour chaque réponse. */
  (*retour)[1].reponses[i] = malloc(14);
}
strcpy((*retour)[1].reponses[0], "int f(0);");
strcpy((*retour)[1].reponses[1], "int f(int 0);");
strcpy((*retour)[1].reponses[2], "int f(int i);");
strcpy((*retour)[1].reponses[3], "int f(i);");
```

```
  (*retour)[1].solution = 3;

  /* QUESTION 2 */
  /* On alloue 40 caractères pour la question. */
  (*retour)[2].question = malloc(40);
  strcpy((*retour)[2].question,
         "Qui pose des questions stupides");

  (*retour)[2].nb_rep = 7;

  for (i = 0; i < (*retour)[2].nb_rep; ++i) {
    /* On alloue 50 caractères pour chaque réponse. */
    (*retour)[2].reponses[i] = malloc(50);
  }
  strcpy((*retour)[2].reponses[0], "le prof. de math");
  strcpy((*retour)[2].reponses[1], "mon copain/ma copine");
  strcpy((*retour)[2].reponses[2], "le prof. de physique");
  strcpy((*retour)[2].reponses[3], "moi");
  strcpy((*retour)[2].reponses[4], "le prof. d'info");
  strcpy((*retour)[2].reponses[5], "personne, il n'y a pas de question stupide");
  strcpy((*retour)[2].reponses[6], "les sondages");

  (*retour)[2].solution = 6;

  return (unsigned int) 3;
}

/* ========================================================================== */
void detruire_examen(Examen* retour)
{
  unsigned int i, j;

  /* Pour chaque question */
  for (i = 0; i < 3; ++i) {

    /* Pour chaque reponse à cette question */
    for (j = 0; j < (*retour)[i].nb_rep; ++j)
    {
      /* On libère la mémoire allouée pour chaque reponse */
      free((*retour)[i].reponses[j]);
    }
    /* On libère la mémoire allouée pour chaque question */
    free((*retour)[i].question);
  }
  /* On libère la mémoire allouée pour l'Examen pointé par retour */
  free(*retour);
  *retour = NULL;
```

```
}
```

Exercice 2 : QCM revisités

Voici une solution.

Par contre, si vous aviez implémenté les tableaux dynamiques de la série passée, c'eût été un bon endroit pour les réutiliser (pour le type `Examen`) !

(fichier src/qcm2.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h> /* pour isspace() */

/* nombre maximum de demandes en cas d'erreur */
#define NB_DEMANDES 3

/* nombre max. de réponses */
#define MAX_REP 10

typedef struct {
  char* question;
  char* reponses[MAX_REP];
  unsigned int nb_rep;
  unsigned int solution;
}  QCM;

typedef QCM* Examen;

void affiche(QCM const * question);
int demander_nombre(int min, int max);
unsigned int poser_question(QCM const * question);
unsigned int creer_examen(Examen*, FILE*);
int demander_fichier(FILE** f);
char* enlever_blancs(char* chaine);

/* ================================================================== */
int main(void)
{
  unsigned int note = 0;
  Examen exam;
  unsigned int taille_examen;
  unsigned int i;
  FILE* fichier;
```

```
  if (! demander_fichier(&fichier)) {
    printf("=> j'abandonne !\n");
    return 1;
  } else {
    taille_examen = creer_examen(&exam, fichier);

    for (i = 0; i < taille_examen; ++i)
      if (poser_question(&(exam[i])) == exam[i].solution)
        ++note;

    printf("Vous avez trouvé %d bonne", note);
    if (note > 1) putchar('s');
    printf(" réponse");
    if (note > 1) putchar('s');
    printf(" sur %d.\n", taille_examen);
  }
  return 0;
}

/* ========================================================================
 * Fonction demander_fichier
 * ------------------------------------------------------------------------
 * In:   Un fichier (par référence) à ouvrir.
 * Out:  Ouvert ou non ?
 * What: Demande à l'utilisateur (au plus NB_DEMANDES fois) un nom de fichier
 *       et essaye de l'ouvrir en lecture.
 * ======================================================================== */
int demander_fichier(FILE** f)
{
  char nom_fichier[FILENAME_MAX+1];
  int taille_lue;
  unsigned short int nb = 0;

  do {
    ++nb;

    /* demande le nom du fichier */
    do {
      printf("Nom du fichier à lire : "); fflush(stdout);
      fgets(nom_fichier, FILENAME_MAX+1, stdin);
      taille_lue = strlen(nom_fichier) - 1;
      if ((taille_lue >= 0) && (nom_fichier[taille_lue] == '\n'))
        nom_fichier[taille_lue] = '\0';
    } while ((taille_lue < 1) && !feof(stdin));

    if (nom_fichier[0] == '\0') {
```

```
      *f = NULL;
      return 0;
    }

    /* essaye d'ouvrir le fichier */
    *f = fopen(nom_fichier, "r");

    /* est-ce que ça a marché ? */
    if (*f == NULL) {
      printf("-> ERREUR, je ne peux pas lire le fichier %s\n",
             nom_fichier);
    } else {
      printf("-> OK, fichier %s ouvert pour lecture.\n",
             nom_fichier);
    }
  } while ((*f == NULL) && (nb < NB_DEMANDES));

  return (*f != NULL);
}

/* ========================================================================== */
void affiche(QCM const * q)
{
  unsigned int i;
  printf("%s ?\n", q->question);
  for (i = 0; i < q->nb_rep; ++i) {
    printf("    %d- %s\n", i+1, q->reponses[i]);
  }
}

/* ========================================================================== */
int demander_nombre(int a, int b)
{
   int res;

   if (a > b) { res=b; b=a; a=res; }

   do {
       printf("Entrez un nombre entier compris entre %d et %d :\n",
              a, b);
       scanf("%d", &res);
    } while ((res < a) || (res > b));
    return res;
}

/* ========================================================================== */
unsigned int poser_question(QCM const * q)
```

```
{
  affiche(q);
  return (unsigned) demander_nombre(1, q->nb_rep);
}

/* ========================================================================== */
unsigned int creer_examen(Examen* retour, FILE* fichier)
{
  QCM* question;
  int erreur = 0;       /* une erreur de format s'est produite */
  int dansquestion = 0; /* en train de lire une question */
#define TAILLE_LIGNE 1025
  char line[TAILLE_LIGNE];      /* ligne à lire */

  unsigned int nb_quest = 0;

  /* taille maximale d'un examen
     Dans une version plus avancée on préfèrera une allocation plus dynamique
     à base de realloc.
   */
#define MAX_QUESTIONS 1000
  *retour = calloc(MAX_QUESTIONS, sizeof(QCM));
  if (*retour == NULL) {
    fprintf(stderr, "Erreur: je ne peux pas allouer %d questions.\n"
            "Recompilez avec un MAX_QUESTIONS plus petit.\n",
            MAX_QUESTIONS);
    return 0;
  }

  /* On fait pointer question sur le premier QCM de l'examen (retour) */
  question = *retour;

  do {
    char *rep;
    rep = fgets(line, TAILLE_LIGNE, fichier); /* lit une ligne */

    if ((rep != NULL) && (line[0] != '#')) {
      /* si c'est bien une ligne et ce n'est pas un commentaire */

      if ((line[0] != 'Q') || (line[1] != ':')) {
        /* si la ligne ne commence pas par "Q:" */

        if (! dansquestion) {
          /* Si on n'a pas encore eu de question : qqchose ne va pas ! */
          fprintf(stderr, "Mauvais format de fichier : pas de \"Q:\"\n");
          erreur = 1;
        } else {
```

```
          /* on a déjà eu une question => c'est donc une ligne de réponse :
             lecture d'une réponse à la question */

          ++(question->nb_rep);
          if (question->nb_rep > MAX_REP) {
            fprintf(stderr, "Je ne peux pas accepter plus que %d réponses\n",
                    MAX_REP);
            erreur = 1;
          }

          else {
            char** current = &(question->reponses[question->nb_rep - 1]);
            if ((line[0] == '-') && (line[1] == '>')) {
              /* la réponse correcte est recopiée */
              *current = malloc(strlen(line)-1); /* remarquez qu'ici strlen(line) est
forcement >= 2 */
              if (*current == NULL) {
                fprintf(stderr,
                        "Erreur: plus de place pour allouer une réponse\n");
                return 0;
              }
              strcpy(*current, &(line[2])); /* supprime le "->" initial */
              if (question->solution != 0) {
                fprintf(stderr,
                        "Hmmm bizard, j'avais déjà une réponse correcte pour"
                        " cette question !\n");
              } else if (enlever_blancs(*current)[0] == '\0') {
                fprintf(stderr,
                        "Hmmm bizard, la réponse indiquée est vide ! "
                        "Cela n'a pas de sens !\n");
                erreur = 1;
              } else {
                question->solution = question->nb_rep;
              }
            } else { /* autre reponse possible */
              if (enlever_blancs(line)[0] == '\0') {
                /* ligne vide -> ignorer */
                --(question->nb_rep);
              } else {
                *current = malloc(strlen(line)+1);
                if (*current == NULL) {
                  fprintf(stderr,
                          "Erreur: plus de place pour allouer une réponse\n");
                  return 0;
                }
                strcpy(*current, line);
              }
```

```
            }
          }
        }
      } else { /* ligne de question : "Q: ..." */
        if (dansquestion) /* passe a la question suivante */
          ++question;

        /* la question est recopiée */
        question->question = malloc(strlen(line)-1);  /* remarquez qu'ici strlen(line)
est forcement >= 2 (line commence par "Q:") */
        strcpy(question->question, &(line[2])); /* recopie sans le "Q:" initial */
        if (enlever_blancs(question->question)[0] == '\0') {
          fprintf(stderr,
                  "Hmmm bizard, la question est vide ! Cela n'a pas de sens !\n");
          erreur = 1;
        } else {
          ++nb_quest;
          question->nb_rep = 0;    /* remets à zéro les réponses... */
          question->solution = 0; /* ...et la solution correcte */
          dansquestion = 1;   /* on a une question */
        }
      }
    }
  } while (!feof(fichier) && !erreur);

  return nb_quest;
}

/* ========================================================================== */
char* enlever_blancs(char* chaine)
{ /* Supprime les blancs initiaux et finaux d'une chaine */

  unsigned char* c1 = NULL;
  unsigned char* c2 = NULL;

  /* Attention ! unsigned est primordial ici pour les caractères >= 128
     (par exemple les accents). Sinon le bit de signe, lors du passage en int,
     va se balader à la mauvaise place !!
     Par exemple isspace('é') renvoie 8 !! */

  /* supprime les blancs finaux */
  for (c1 = (unsigned char *) (chaine + strlen(chaine) - 1);
       (c1 >= (unsigned char *) chaine) && isspace(*c1); --c1)
    *c1 = '\0';

  /* supprime les blancs initiaux */
  if (isspace((unsigned char) chaine[0])) {
```

```
    for (c1 = (unsigned char *) chaine; *c1 && isspace(*c1); ++c1);
    for (c2 = c1; *c2; ++c2)
      chaine[c2-c1] = *c2;
    chaine[c2-c1] = '\0'; }

  return chaine;
}
```

Exemple de fichier d'examen :

```
Q:Combien de dents possède un éléphant adulte
32
-> de 6 à 10
beaucoup
24
2

Q: Laquelle des instructions suivantes est un prototype de fonction
int f(0);
int f(int 0);
-> int f(int i);
int f(i);

Q:Qui pose des questions stupides
le prof. de math
mon copain/ma copine
le prof. de physique
moi
le prof. d'info
->personne, il n'y a pas de question stupide
les sondages

Q: Quel signe est le plus étrange
 #
 ->
->->##<-
#b
a
```

Exercice 3 : Listes chaînées

Voir le cours pour l'essentiel de cet exercice. Voici le code complet :

(fichier src/listechainee.c)

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

```
/* deux definition utiles */
#define LISTE_VIDE (NULL)
#define est_vide(L) ((L) == LISTE_VIDE)

/* deux definitions pour changer facilement de type */
#define type_el int
#define affiche_el(T) printf("%d", T)

/* la structure de liste chainee */
typedef struct Element_ Element;
typedef Element* ListeChainee;
struct Element_ {
    type_el valeur;
    ListeChainee suite;
};

/* les "méthodes" */
void insere_entete(ListeChainee* liste, type_el une_valeur);
void insere_apres(Element* existant, type_el a_inserer);
void supprime_tete(ListeChainee* liste);
void supprime_suivant(Element* e);
size_t taille(ListeChainee liste);
void affiche_liste(ListeChainee liste);

/* ============================================================================ */
void insere_entete(ListeChainee* liste, type_el une_valeur)
{
  const ListeChainee tmp = *liste;

  *liste = malloc(sizeof(Element));
  if (*liste != NULL) {
    (*liste)->valeur = une_valeur;
    (*liste)->suite = tmp;
  } else {
    /* si on n'a pas pu faire d'allocation, au moins on restitue
       l'origine.
       Mieux : il faudrait l'indiquer par une valeur de retour */
    *liste = tmp;
  }
}

/* ============================================================================ */
void insere_apres(Element* existant, type_el a_inserer)
{
  Element* e;
  e = malloc(sizeof(Element));
```

```
  if (e != NULL) {
    e->valeur = a_inserer;
    e->suite = existant->suite;
    existant->suite = e;
  }
}

/* ========================================================================== */
void supprime_tete(ListeChainee* liste)
{
  if (!est_vide(*liste)) {
    ListeChainee nouvelle = (*liste)->suite;
    free(*liste);
    *liste = nouvelle;
  }
}

/* ========================================================================== */
void supprime_suivant(Element* e)
{
  /* supprime le premier élément de la liste "suite" */
  supprime_tete(&(e->suite));
}

/* ========================================================================== */
size_t taille(ListeChainee liste)
{
  size_t t = 0;
  while (!est_vide(liste)) {
    ++t;
    liste = liste->suite;
  }
  return t;
}

/* ========================================================================== */
void affiche_liste(ListeChainee liste)
{
    ListeChainee i;
    putchar('(');
    for (i = liste; !est_vide(i); i = i->suite) {
      affiche_el(i->valeur);
      if (!est_vide(i->suite)) printf(", ");
        }
    putchar(')');
}
```

```
/* ========================================================================== */
int main(void) {
  ListeChainee maliste = LISTE_VIDE;
  type_el un_element = 3;

  printf("J'insère (en tête) "); affiche_el(un_element); putchar('\n');
  insere_entete(&maliste, un_element);

  printf("J'insère (en tête) 2, puis 1, 0 et -1.\n");
  insere_entete(&maliste, 2);
  insere_entete(&maliste, 1);
  insere_entete(&maliste, 0);
  insere_entete(&maliste, -1);

  printf("Voici la liste : \n\t");
  affiche_liste(maliste); putchar('\n');

  printf("Je supprime la tete de liste.\n");
  supprime_tete(&maliste);
  printf("Voici la liste : \n\t");
  affiche_liste(maliste); putchar('\n');

  printf("Je supprime le 4e element.\n");
  supprime_suivant(maliste->suite->suite);
  printf("Voici la liste : \n\t");
  affiche_liste(maliste); putchar('\n');

  printf("J'insere 156 en 3e position.\n");
  insere_apres(maliste->suite, 156);
  printf("Voici la liste : \n\t");
  affiche_liste(maliste); putchar('\n');

  printf("de taille %u.\n", taille(maliste));

  return 0;
}
```

Pour éviter les parcours infini sur les listes cycliques, le meilleur moyen est de modifier la définition de la liste elle-même en ajoutant un indicateur (par exemple un `char`) marquant si on est déjà passé par l'élément courant ou non.
On parcourt alors la liste tant que l'indicateur de l'élément courant n'indique pas qu'on est déjà passé. Il faut bien sur changer cet indicateur à chaque passage. Il faut de plus prévoir une fonction qui remet à zéro ces indicateurs... elle même ne devant pas boucler infiniment.

Exercice 4 : Carnet d'adresses

(fichier src/carnet.c)

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

#define MAXARGS 10

#define MAXSTR  32
#define MAXLINE 256

/* un noeud de l'arbre */
typedef struct addr__ addr_t;
struct addr__ {
    addr_t *left;               /* noeud gauche */
    addr_t *right;              /* noeud droite */

    char name[MAXSTR];  /* clé du noeud */
    char num [MAXSTR];  /* valeur du noeud */

    /*  valeurs supplémentaires ici */
};

/* l'arbre, défini par sa racine */
typedef struct book__ {
    addr_t* root;
} book_t;

/* ------------------------------------------------- */

/* créer un nouveau carnet d'adresses vide */
book_t* book_create(void);

/* libérer les ressources associées */
void book_free(book_t* b);

/* ajouter ou modifier une entrée du carnet d'adresse */
void book_add(book_t* b, const char* name, const char* num);

/* supprimer une entrée du carnet d'adresses */
void book_remove(book_t* b, const char* name);

/* lister dans l'ordre tous les noms du carnet d'adresses */
void book_list(book_t* b);

/* afficher une entrée du carnet d'adresses */
void book_view(book_t* b, const char* name);
```

```
/* remplacer le contenu du carnet d'adresses par celui du fichier */
void book_load(book_t* b, const char* file);

/* sauver le contenu du carnet dans un fichier au format CSV */
void book_save(book_t* b, const char* file);

/*
 * Note: le format CSV est: un enregistrement par ligne, où les
 * champs sont séparés par des ';'. Par exemple:
 *
 * nom1;numéro1;\n
 * nom2;numéro2;\n
 * ...
 *
 */

/* ------------------------------------------------- */

/*
 * donne l'adresse du pointeur qui conduit à l'enregistrement
 * - si l'enregistrement n'existe pas, retourne l'adresse du pointeur
 *   à modifier pour rajouter l'enregistrement
 * - si l'enregistrement existe, retourne l'adresse du pointeur
 *   vers l'enregistrement
 */
addr_t** book_find(book_t*, const char* name);

/* manipulateurs d'adresse (notamment pour la récursivité */
addr_t* addr_create(const char* name, const char* num);
void addr_free(addr_t* a);
void addr_list(addr_t* a);
addr_t* addr_read(FILE* f);
void addr_write(addr_t* a, FILE* f);

/* ================================================================== */


/* copie <src> dans <dst> qui a la taille <size>, tronque si besoin */
#if !defined(__APPLE__) && !defined(__OpenBSD__) && !defined(__FreeBSD__)
size_t strlcpy(char* dst, const char* src, size_t size)
{
    size_t len = strlen(src);
    size_t real = (len>size-1) ? size-1 : len;
    strncpy(dst, src, real);
    dst[real] = 0;
    return len;
```

```
}
#endif

/* ------------------------------------------------- */

addr_t* addr_create(const char* name, const char* num)
{
    addr_t* a = malloc(sizeof(addr_t));
    if (!a) return NULL;
    a->left=NULL;
    a->right=NULL;
    strlcpy(a->name, name, MAXSTR);
    strlcpy(a->num, num, MAXSTR);
    return a;
}

void addr_list(addr_t* a)
{
    if (a->left) addr_list(a->left);
    printf("  - %s\n",a->name);
    if (a->right) addr_list(a->right);
}

void addr_free(addr_t* a)
{
    if (a==NULL) return;
    if (a->left) addr_free(a->left);
    if (a->right) addr_free(a->right);
    free(a);
}

addr_t* addr_read(FILE* f)
{
    char buf[MAXLINE];
    if (fgets(buf,MAXLINE,f)) {
        char* num=NULL;
        char* p;
        if ((p=strchr(buf,'\n'))) *p=0;
        if ((p=strchr(buf,';'))) {
            *p=0;
            num = &p[1];
            if ((p=strchr(num,';'))) *p=0;
            return addr_create(buf,num);
        }
    }
    return NULL;
}
```

```
void addr_write(addr_t* a, FILE* f)
{
    if (a->left) addr_write(a->left, f);
    fprintf(f,"%s;%s;\n",a->name,a->num);
    if (a->right) addr_write(a->right, f);
}

/* ------------------------------------------------- */

book_t* book_create(void)
{
    book_t* b = malloc(sizeof(book_t));
    if (b==NULL) return NULL;
    b->root=NULL;
    return b;
}

void book_free(book_t* b)
{
    if (b==NULL) return;
    if (b->root!=NULL) addr_free(b->root);
    free(b);
}

addr_t** book_find(book_t* b, const char* name)
{
    int found = 0; /* si un enregistrement existe déjà */
    addr_t **pp = &b->root; /* pointeur vers le pointeur qui référence l'adresse */

    /* tant qu'on ne trouve pas un pointeur NULL (une feuille)
     * ou la valeur qu'on cherche, on parcourt la structure */
    while((*pp!=NULL) && found==0) {
        addr_t *a = *pp; /* on récupère un pointeur vers l'adresse référencée */
        int cmp = strcmp(a->name, name);
        if (cmp>0) pp =  &(a->left);
        else if (cmp<0) pp = &(a->right);
        else found=1; /* pp pointe a */
    }
    return pp;
}

void book_add(book_t* b, const char* name, const char* num)
{
    addr_t **pp = book_find(b, name);
    if (*pp != NULL) {
        addr_t *a = *pp;
```

```
        strlcpy(a->num, num, MAXSTR); /* une adresse existe, on la met à jour */
    } else {
        addr_t *a = addr_create(name, num);
        if (a != NULL) *pp = a; /* on crée l'adresse qu'on attache au pointeur */
    }
}

void book_remove(book_t* b, const char* name)
{
    addr_t **pp = book_find(b, name);
    if (*pp == NULL) printf("Pas trouvé.\n");
    else {
        addr_t *a = *pp; /* on prend l'enregistrement */
        *pp = NULL; /* on la déconnecte de l'arbre */
        /* on remet les enfants dans l'arbre */
        if (a->left)  {
            pp = book_find(b, a->left->name );
            *pp = a->left ;
        }
        if (a->right) {
            pp = book_find(b, a->right->name);
            *pp = a->right;
        }
        free(a);
    }
}

void book_list(book_t* b)
{
    if (b->root) addr_list(b->root);
    else printf("le carnet d'adresses est vide.\n");
}

void book_view(book_t* b, const char* name)
{
    addr_t **pp = book_find(b, name);
    if (*pp==NULL) printf("Pas trouvé.\n");
    else {
        addr_t *a = *pp; /* on prend l'enregistrement */
        printf("Vous pouvez appeler %s au numéro %s.\n", a->name, a->num);
    }
}

void book_load(book_t* b, const char* file)
{
    char fname[MAXLINE];
    snprintf(fname, MAXLINE, "%s.csv", file);
```

```
    FILE* f = fopen(fname, "r");
    if (f != NULL) {
        /* on vide le carnet d'adresses */
        if (b->root) {
            addr_free(b->root);
            b->root=NULL;
        }

        /* on lit les adresses */
        addr_t *a;
        while ((a=addr_read(f))) {
            addr_t **pp = book_find(b, a->name);
            *pp = a;
        }

        fclose(f);
    } else {
        printf("Impossible de lire %s.\n",fname);
    }
}

void book_save(book_t* b, const char* file)
{
    char fname[MAXLINE];
    snprintf(fname, MAXLINE, "%s.csv", file);
    FILE* f = fopen(fname, "w");
    if (f != NULL) {
        if (b->root) addr_write(b->root, f);
        fclose(f);
    } else {
        printf("Impossible d'écrire %s.\n",fname);
    }
}

int main(void)
{
    int quit = 0;
    book_t *b = book_create();

    /* données de test */
    book_add(b,"Lucien"  ,"012 345 67 89");
    book_add(b,"Stéphane","021 879 51 32");
    book_add(b,"Julien"  ,"079 523 12 45");
    book_add(b,"Antoine" ,"076 125 08 78");
    book_add(b,"Damien"  ,"022 329 08 85");

    while (!quit) {
```

```
        /* on lit la commande dans un buffer */
        char cmd[MAXSTR*2];
        printf("> ");
        fflush(stdout);
        char* check = fgets(cmd,MAXSTR*2,stdin);

        if ((check != NULL) && (cmd[0] != '\n')) {

            /* on découpe la commande en arguments, voir man strsep  */
            int  an = 0;        /* nombre d'arguments  */
            char* a[MAXARGS];   /* tableau d'arguments */
            char* ptr = cmd;
            while (an<MAXARGS && ((a[an]=strtok(ptr," \t\n")) != NULL)) {
                if (a[an][0]!='\0') ++an;
                ptr = NULL;
            }

            if (a[0] != NULL) {
                /* on interprète la commande */
                if (!strcmp(a[0],"add") && an>2) book_add(b,a[1],a[2]);
                else if (!strcmp(a[0],"del") && an>1) book_remove(b,a[1]);
                else if (!strcmp(a[0],"view")) book_view(b,a[1]);
                else if (!strcmp(a[0],"list")) book_list(b);
                else if (!strcmp(a[0],"load") && an>1) book_load(b,a[1]);
                else if (!strcmp(a[0],"save") && an>1) book_save(b,a[1]);
                else if (!strcmp(a[0],"quit")) {
                    printf("Au revoir.\n");
                    quit=1;
                } else if (!strcmp(a[0],"help")) {
                    printf("  add <name> <num>  ajouter un numéro\n");
                    printf("  del <name>        supprimer un numéro\n");
                    printf("  view <name>       afficher les informations\n");
                    printf("  list              lister les noms\n");
                    printf("  load <file>       lit les adresses du fichier\n");
                    printf("  save <file>       enregistre les adresses dans le
fichier\n");
                    printf("  quit              quitter le programme\n");
                } else printf("Commande erronnée, entrez 'help' pour l'aide.\n");
            }
        }
    }
    book_free(b);
    return 0;
}
```

Exercice 5 : Dérivation formelle

Ce corrigé gagnerait à être modularisé (compilation séparée), mais le sujet n'ayant pas encore été abordé (semaine 11), la solution est ici proposée en un seul fichier.

(fichier src/derivation/bigone.c)

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

/* ========================================================================
 * ARBRES - Data structures et Prototypes
 * ======================================================================== */

typedef struct arbre_ Arbre;
struct arbre_ {
  char valeur;
  Arbre* gauche;
  Arbre* droite;

  /* necessaire si pointages multiples possibles, ie si le même sous-arbre
     est utilisé par différents noeuds pères, par exemple dans des arbres
     différents.
     Une alternative serait d'utiliser des deep copies dans creer_arbre au
     lieu de faire pointer tout le monde au même endroit, mais cette
     dernière solution est plus lourde.
  */
  unsigned int nb_acces;
};

Arbre* creer_arbre(char valeur, Arbre* g, Arbre* d);
void affiche_arbre(Arbre* a);
void libere_arbre(Arbre** a);

/* ========================================================================
 * ARBRES - Definitions
 * ======================================================================== */

Arbre* creer_arbre(char valeur, Arbre* g, Arbre* d)
{
  Arbre* arbre;

  arbre = malloc(sizeof(Arbre));
  if (arbre == NULL) {
    fprintf(stderr, "Erreur : plus assez de mémoire pour faire pousser "
            "un nouvelle arbre\n");
  } else {
    arbre->valeur = valeur;
    arbre->gauche = g;
```

```
    if (g != NULL) ++(g->nb_acces);
    arbre->droite = d;
    if (d != NULL) ++(d->nb_acces);
    arbre->nb_acces = 0;
  }

  return arbre;
}

void affiche_arbre(Arbre* a)
{
  if (a != NULL) {
    if (a->gauche != NULL) {
      putchar('(');
      affiche_arbre(a->gauche);
      putchar(' ');
    }
    printf("%c", a->valeur);
    if (a->droite != NULL) {
      putchar(' ');
      affiche_arbre(a->droite);
      putchar(')');
    }
  }
}

void libere_arbre(Arbre** pa)
{
  if (pa != NULL) {
    Arbre* const a = *pa; // juste pour simplifier l'écriture
    if (a != NULL) {
      if (a->nb_acces > 0) { --(a->nb_acces); }
      if (a->nb_acces == 0) {
        libere_arbre(&(a->gauche));
        libere_arbre(&(a->droite));
        free(a);
        *pa = NULL; /* pas "a" ici, bien sûr ! */
      }
    }
  }
}

/* ========================================================================
 * DERIVATION
 * ======================================================================== */

Arbre* derive(Arbre* arbre, char variable)
```

```
{
  Arbre* resultat;
  Arbre* derive_gauche;
  Arbre* derive_droite;

  if (arbre == (Arbre*) NULL)
    return arbre;

  /* Pre-calcul des derives a gauche et a droite */
  derive_gauche = derive(arbre->gauche, variable);
  derive_droite = derive(arbre->droite, variable);

  switch (arbre->valeur) {
  case '+':
  case '-':
    resultat = creer_arbre(arbre->valeur, derive_gauche, derive_droite);
    break;

  case '*':
      resultat =
        creer_arbre('+',
                    creer_arbre('*', derive_gauche, arbre->droite),
                    creer_arbre('*', arbre->gauche, derive_droite)
                    );
    break;

  case '/':
      resultat =
        creer_arbre('/',
                    creer_arbre('-',
                                creer_arbre('*', derive_gauche, arbre->droite),
                                creer_arbre('*', arbre->gauche, derive_droite)),
                    creer_arbre('*', arbre->droite, arbre->droite)
                    );
      break;

  case '^':
    resultat =
      creer_arbre('*',
                  creer_arbre('*', arbre->droite, derive_gauche),
                  creer_arbre('^', arbre->gauche,
                              creer_arbre('-',
                                          arbre->droite,
                                          creer_arbre('1',NULL,NULL)))
                  );
    libere_arbre(&derive_droite); // on ne fait pas de f(x)^g(x) ;-)
    break;
```

```
  default:
    if (arbre->valeur == variable)
      resultat = creer_arbre('1', NULL, NULL);
    else
      resultat = creer_arbre('0', NULL, NULL);
  }

  return resultat;
}

/* ================================================================
 * MAIN
 * ================================================================ */

int main(void)
{
  Arbre *a = creer_arbre('a', NULL, NULL);
  Arbre *b = creer_arbre('b', NULL, NULL);
  Arbre *x = creer_arbre('x', NULL, NULL);
  Arbre *d_expr1, *d_expr2, *d_expr3,*d_expr4;

  Arbre *xpa = creer_arbre('+', x, a);
  Arbre *expr1 = xpa;
  Arbre *xpb = creer_arbre('+', x, b);
  Arbre *x2  = creer_arbre('*', x, x);

  Arbre *expr2 = creer_arbre('*', xpa, xpb);
  Arbre *expr3 = creer_arbre('+', creer_arbre('*', x, x2),
                             creer_arbre('*', a, x));
  Arbre *expr4 = creer_arbre('/', creer_arbre('^', x, a),
                             creer_arbre('+', x2, creer_arbre('*', b, x)));

  affiche_arbre(expr1); printf("\n");
  affiche_arbre(expr2); printf("\n");
  affiche_arbre(expr3); printf("\n");
  affiche_arbre(expr4); printf("\n");

  /* Calcul des derivees */
  d_expr1 = derive(expr1,'x');
  d_expr2 = derive(expr2,'x');
  d_expr3 = derive(expr3,'x');
  d_expr4 = derive(expr4,'x');

  /* Afficher les derivees */
  printf("d(expr1)/dx = ");
  affiche_arbre(d_expr1);
```

```
  printf("\nd(expr2)/dx = ");
  affiche_arbre(d_expr2);
  printf("\nd(expr3)/dx = ");
  affiche_arbre(d_expr3);
  printf("\nd(expr4)/dx = ");
  affiche_arbre(d_expr4);
  printf("\n");

  /* liberation de la memoire */
  libere_arbre(&d_expr1);
  libere_arbre(&d_expr2);
  libere_arbre(&d_expr3);
  libere_arbre(&d_expr4);
  libere_arbre(&expr2);
  libere_arbre(&expr3);
  libere_arbre(&expr4);

  return 0;
}
```

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