Correction series 03: Pointers 2

Exercise 1: Automatic letter generator

First version of the code:

(file lettre1.c)

```c
#include <stdio.h>

void genereLettre(void)
{
  printf(
    "Bonjour chère Mireille,\n"
    "Je vous écris à propos de votre cours.\n"
    "Il faudrait que nous nous voyons le 18/12 pour en discuter.\n"
    "Donnez-moi vite de vos nouvelles !\n"
    "Amicalement, John.\n"
  );
}

int main(void)
{
  genereLettre();
  return 0;
}
```

Seconde version of the code:

(file lettre2.c)

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

 typedef enum { MASCULIN, FEMININ } Genre;

void genereLettre(Genre genre, const char* destinataire, const char* sujet,
          unsigned int jour, unsigned int mois, const char* politesse,
          const char* auteur)
{
  printf("Bonjour ");
  if (genre == MASCULIN)
    printf("cher");
  else
    printf("chère");
  printf(" %s,", destinataire);

  printf(
    "Je vous écris à propos de %s.\n"
    "Il faudrait que nous nous voyons le %d/%d pour en discuter.\n"
    "Donnez-moi vite de vos nouvelles !\n"
    "%s, %s.\n"
    , sujet, jour, mois, politesse, auteur);
}

int main(void)
{
  genereLettre(FEMININ, "Mireille", "votre cours" , 18, 12, "Amicalement",
           "John");
  putchar('\n');
  genereLettre(MASCULIN, "John", "votre demande de rendez-vous", 16, 12,
           "Sincèrement", "Mireille");
  return 0;
}
```

Notes:

1. Using the `char*` type in the function prototype allows you to pass character strings as parameters, as illustrated in the `main`.
2. We use `const` for these arguments here, because this function does not modify their value. This also avoids the risk of mishandling literal constants.

Exercise 2: Word segmentation

(file token.c)

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

/* La fonction suivante teste si le caractère est un séparateur
 *
 * Écrire une fonction présente l'avantage de pouvoir redéfinir facilement
 * la notion de séparateur (et éventuellement d'en définit plusieurs)
 */
int issep (char c) {
  return (c == ' '); /* retourne 1 si la condition est vérifiée, 0 sinon */
}

/* Il y a *plein* d'autres façons d'écrire cette fonction.
 *
 * Je trouve celle-ci élégante.
 * Elle le serait encore plus en utilisant « l'arithmétique des pointeurs »
 * (présentée en semaine 9). Je vous conseille donc de reprendre cet exercice
 * (et changer le prototype de cette fonction) en semaine 9.
 */
int nextToken(char const* str, size_t* from, size_t* len)
{
  const size_t taille = strlen(str); // taille totale de la ligne entrée

  /* D'abord, on saute tous les séparateurs avant le premier
   * mot à partir de from.
   * Notez que *from représente la valeur pointée par from,
   * càd l'index qu'on a donné en paramètre.
   */
  while ( (*from < taille) && issep(str[*from]) ) {
    ++(*from); // on veut incrémenter la valeur pointée par from, pas son adresse !
  }

  /* Maintenant, from pointe sur l'index de la première lettre
   * du premier mot qui nous intéresse.
   * On avance jusqu'au prochain séparateur ou la fin de str.
   */
  *len = 0;
  for (size_t i = *from; ((i < taille) && !issep(str[i])); ++(*len), ++i);

  return (*len != 0);
}

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

/* On définit une TAILLE_MAX pour la phrase qui sera entrée par
   l'utilisateur. */
#define TAILLE_MAX 1024

int main(void)
{
  char phrase[TAILLE_MAX+1] = "";
```

```
  printf("Entrez une chaîne : ");
  fgets(phrase, TAILLE_MAX+1, stdin);
  if (phrase[0] != '\0') {
    /* On supprime le '\n' lu, c.-à-d. le dernier caractère
     * (en le remplaçant par '\0')
     */
    const size_t taille_lue = strlen(phrase) - 1;
    if (phrase[taille_lue] == '\n') phrase[taille_lue] = '\0';
  }

  size_t debut    = 0;
  size_t longueur = 0;
  printf("Les mots de \"%s\" sont :\n", phrase);
  while (nextToken(phrase, &debut, &longueur)) {

    /* On copie dans la variable mot les caractères
     * de la phrase depuis l'index debut,
     * de la taille longueur.
     */
    char mot[TAILLE_MAX+1];
    strncpy(mot, &(phrase[debut]), longueur);

    /* On rajoute un '\0' pour indiquer la fin du mot
     * (sinon on ne peut pas employer %s).
     */
    mot[longueur] = '\0'; /* fin de mot */
    printf("'%s'\n", mot);
    debut += longueur;
  }
  return 0;
}
```

Exercise 3: Integrals revisited

(file integrale2.c)

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

double f1(double x) { return x*x;             }
double f2(double x) { return sqrt(exp(x));    }
double f3(double x) { return log(1.0+sin(x)); }

typedef double (*Fonction)(double);

double demander_nombre(void)
{
  double res;
  printf("Entrez un nombre réel : ");
  scanf("%lf", &res);
  return res;
}

Fonction demander_fonction(void)
{
  int rep;
  Fonction choisie;
  do {
    printf("De quelle fonction voulez vous calculer l'intégrale [1-5] ? ");
    scanf("%d", &rep);
  } while ((rep < 1) || (rep > 5));
```

```
    switch (rep) {
    case 1: choisie = f1 ; break ;
    case 2: choisie = f2 ; break ;
    case 3: choisie = f3 ; break ;
    case 4: choisie = sin ; break ;
    case 5: choisie = exp ; break ;
    }

  return choisie;
}

double integre(Fonction f, double a, double b)
{
  double res =
    41.0 * ( f(a) + f(b) )
      + 216.0 * ( f((5*a+b)/6.0) + f((5*b+a)/6.0) )
      +  27.0 * ( f((2*a+b)/3.0) + f((2*b+a)/3.0) )
      + 272.0 * f((a+b)/2.0) ;
  res *= (b-a) / 840.0;
  return res;
}

int main(void)
{
  const double a = demander_nombre();
  const double b = demander_nombre();
  const Fonction choix = demander_fonction();

  printf("Integrale entre %f et %f : %f\n", a, b,
         integre(choix, a, b));
  return 0;
}
```

More advanced version (level 3, not requested)

(file integrale3.c)

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

double f1(double x) { return x*x;             }
double f2(double x) { return sqrt(exp(x));    }
double f3(double x) { return log(1.0+sin(x)); }

typedef double (*Fonction)(double);

typedef struct {
  const char* nom;
  Fonction f;
} Choix_fonction;

double demander_nombre(const char* msg)
{
  printf("Entrez un nombre réel (%s) : ", msg);
  double res = 0.0;
  scanf("%lf", &res);
  return res;
}

size_t demander_fonction(Choix_fonction* choix, size_t nb)
{
  printf("De quelle fonction voulez vous calculer l'intégrale :\n");
  for (size_t i = 0; i < nb; ++i) {
```

```
    printf("  %zu. %s\n", i + 1, choix[i].nom);
  }
  puts("?");
  size_t rep = 0;
  do {
    scanf("%zu", &rep);
  } while ((rep < 1) || (rep > nb));
  return rep - 1;
}

double integre(Fonction f, double a, double b)
{
  double res =
    41.0 * ( f(a) + f(b) )
      + 216.0 * ( f((5*a+b)/6.0) + f((5*b+a)/6.0) )
      +  27.0 * ( f((2*a+b)/3.0) + f((2*b+a)/3.0) )
      + 272.0 * f((a+b)/2.0) ;
  res *= (b-a) / 840.0;
  return res;
}

int main(void)
{
  Choix_fonction choix[] = {
    { "x^2"           , f1  },
    { "sqrt(exp())"   , f2  },
    { "log(1 + sin())", f3  },
    { "sinus"         , sin },
    { "exp()"         , exp }
  };

  const double a = demander_nombre("borne basse, a");
  const double b = demander_nombre("borne haute, b");
  const size_t n = demander_fonction(choix, sizeof(choix) / sizeof(choix[0]));

  printf("Integrale entre %f et %f de %s : %f\n", a, b,
         choix[n].nom, integre(choix[n].f, a, b));

  return 0;
}
```

Exercise 4 : mini VM

(file interp_cmd.c)

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

typedef void (*Cmd)(void* data);

// Notre machine simpliste
void print(void* data);
void add(void* data);
void push(void* data);
void pop(void* data);
void quit(void* data);

Cmd interprete(const char* nom_commande);

// ----------------------------------------------------------------------
int main(void)
{
```

```
  double registres[] = { 0.0, 0.0 };

  Cmd cmd = quit;
  do {
    char lu[] = "nom de la plus longue commande";
    printf("Entrez une commande (print, add, push, pop, quit) : ");
    scanf("%s", lu);
    (cmd = interprete(lu))(registres);
  } while (cmd != quit);

  return 0;
}

// ----------------------------------------------------------------------
Cmd interprete(const char* nom)
{
  if (nom == NULL) return quit;

  if ( ! strcmp(nom, "print") ) {
    return print;
  } else
  if ( ! strcmp(nom, "add") ) {
    return add;
  } else
  if ( ! strcmp(nom, "push") ) {
    return push;
  } else
  if ( ! strcmp(nom, "pop") ) {
    return pop;
  }
  return quit;
}

// ----------------------------------------------------------------------
void print(void* data)
{
  const double * const px = data;
  printf("-> %g\n", *px);
}

// ----------------------------------------------------------------------
void add(void* data)
{
  double * const regs = data;
  regs[0] += regs[1];
}

// ----------------------------------------------------------------------
void push(void* data)
{
  double * const regs = data;
  regs[1] = regs[0];
  printf("Valeur ? ");
  scanf("%lf", regs);
}

// ----------------------------------------------------------------------
void pop(void* data)
{
  double * const regs = data;
  regs[0] = regs[1];
}
```

```
// ------------------------------------------------------------------------
void quit(void* useless)
{
  puts("Bye!");
}
```

Exercise 5: linked list

(file 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;
}
```

The best way to avoid infinite traversals on cyclic lists is to modify the definition of the list itself by adding an indicator (for example a `char`) marking whether we have already passed through the current element or not.
We then go through the list as long as the indicator of the current element does not indicate that we have already gone through. You must of course change this indicator each time you pass. It is also necessary to provide a function which resets these indicators to zero itself not having to loop infinitely.

Exercise 6: return to the memory explorer

(file 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 7 : QCM

(file 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;
}
```

Exercise 7: MCQ questionnaire

Here is a solution.
On the other hand, if you had implemented the dynamic arrays from the past series, this would have been a good place to reuse them (for the `Exam` type)!

(file 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 1024
  char line[TAILLE_LIGNE+1];      /* 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+1, 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
        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;
}
```

Example of exam sheet:

```
Q:How many teeth does an adult elephant have
32
          -> from 6 to 10

24
2

Q:Which of the following statements is a function prototype
int f(0);
int f(int 0);
          -> int f(int i);
int f(i);

Q:Who asks stupid questions
the math prof.
my boyfriend/girlfriend
```

```
the physics prof.
me
the programming prof.
->no one, there are no stupid questions
polls

Q:Which sign is the strangest
#
          ->
          ->->##<-
#b
a
```

Exercise 9: Address book

(file carnet.c)

```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);
```

```c
/* 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;
}
```

```c
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;
}
```

Exercise 10 : Needleman-Wunsch (a.k.a. Viterbi)

(file needleman-wunsch.c)

```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;
```

```c
/* 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;
}

#define max(x, y) ((x) > (y) ? (x) : (y))
Solution* extractSolution(const Table* tab)
{
    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) {
        free(sol);
        return NULL;
    }

    size_t i = tab->height ? (tab->height - 1) : 0;
    size_t 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: //continue
        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: //continue
        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;
}
```

Exercise 11: Formal derivation

This correction would benefit from being modularized (separate compilation), but as the subject has not yet been addressed, the solution is proposed here in a single large file.

(file 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
 * ============================================================================ */
```

```c
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;
}
```