# Correction series 2 : Pointers 1

## Exercise 1: Exploring memory (level 2)

Here I present two variations of displaying the bits of a byte. There are of course many more! It is mainly to illustrate from a practical point of view little-commented operations in progress (binary operations on memory).

One of the solutions also presents the ternary operator `?:`: “`A? B: C`” is similar to “`if (A) { B } else { C }`”.

(file memory_view.c)

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

typedef unsigned char octet;

// ==================================================================
// version 1

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);
}

/* version 2 : moins bonne que ci-dessus :
 * affiche les bits « à l'envers » et n'affiche
 * pas les 0 de poids fort.
 */
void affiche_binaire_2(octet c) {
  do {
    if (c & 1) putchar('1');
    else       putchar('0');
    c >>= 1; // ou c /= 2;
  } while (c);
}

// ==================================================================
void affiche(size_t i, octet c) {
  printf("%02zu : ", i);
  affiche_binaire(c);
  printf(" %3u ", (unsigned int) c);
  if ((c >= 32) && (c <= 126)) {
    printf("'%c'", c);
  }
  putchar('\n');
}

// ==================================================================
void dump_mem(const octet* ptr, size_t length)
{
  /* solution simple qui pourra être améliorée
   * lorsque nous aurons vu l'arithmétique des pointeurs
   */
  printf("A partir de %p :\n", ptr);
  for (size_t i = 0; i < length; ++i) {
    affiche(i, ptr[i]);
  }
}
```

```c
// ======================================================================
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;
}
```

---

## Exercise 2: dynamic arrays

(file vector.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h> // pour SIZE_MAX
#include <string.h> // pour memset

#define VECTOR_PADDING 32
#define TYPE int

typedef struct {
  size_t size;      // nombre d'éléments utilisés dans le tableau
  size_t allocated; // nb élements déjà alloués
  TYPE* content;    // tableau de contenu (alloc. dyn.)
} vector;

// ======================================================================
vector* construct_vector(vector* v) {
  vector* result = v;
  if (result != NULL) {
    result->size = 0;
    result->content = calloc(VECTOR_PADDING, sizeof(TYPE));
    if (result->content != NULL) {
      result->allocated = VECTOR_PADDING;
    } else {
      result->allocated = 0;
      result = NULL;
    }
  }
  return result;
}

// ======================================================================
void destruct_vector(vector* v) {
  if (v != NULL) {
    if (v->content != NULL) {
      free(v->content);
      v->content = NULL;
      v->size = v->allocated = 0;
    }
  }
}

// ======================================================================
/* Notez bien la différence entre construct_vector(), qui prend un vector
 * par référence et la construit (= l'initialise),
 * et ici create_vector() qui alloue dynamiquement un vector, LUI-MÊME
 * (et non pas que son contenu !).
 * Exemples d'utilisation :
 *
 *    vector v; // le vector existe
 *    if (construct_vector(&v) != NULL)  // passage par référence
 *
 *    vector* pv = NULL; // il n'y a ici aucun vector qui existe
 *    ...
 *    pv = create_vector();
 *    if (pv != NULL) ...
 */

vector* create_vector(void) {
  vector* v = malloc(sizeof(vector));
  if (v != NULL) {
    if (construct_vector(v) == NULL) {
      free(v);
      v = NULL;
    }
  }
  return v;
}

// ======================================================================
void delete_vector(vector** v) {
  if (*v != NULL) {
    destruct_vector(*v);
    free(*v);
    *v = NULL;
  }
}

// ======================================================================
vector* empty_vector(vector* v) {
  if (v != NULL) {
      v->size = 0;
      // réinitialisation à 0 de tout le contenu (évite des fuites d'information)
      memset(v->content, 0, v->allocated * sizeof(TYPE));
      /* Notez qu'ici la multiplication par sizeof(TYPE) NE peut PAS déborder
       * car déjà vérifiée lors de enlarge_vector().                          */
  }
  return v;
}

// ======================================================================
vector* enlarge_vector(vector* v) {
  if (v != NULL) {
    vector result = *v;
    result.allocated += VECTOR_PADDING;
    if ((result.allocated > SIZE_MAX / sizeof(TYPE)) ||
        ((result.content = realloc(result.content,
                                   result.allocated * sizeof(TYPE)))
         == NULL)) {
      return NULL; /* retourne NULL en cas d'échec ;
                    * v n'a pas été modifié.
                    */
    }

    // initialisation à 0 de la nouvelle partie allouée
    memset( &(result.content[v->allocated]), 0, VECTOR_PADDING * sizeof(TYPE));
    // plus tard, on écrira « result.content + v->allocated » au lieu de « &(result.content[v->allocated])

    // affectation finale, tout d'un coup (opération atomique)
    *v = result;
}
return v;}

// ====================================================================
int ensure_capacity(vector* vect) {
  if (vect != NULL) {
    while (vect->size >= vect->allocated) {
      if (enlarge_vector(vect) == NULL) {
        return 0;
      }
    }
    return 1;
  }
  return 0;
}

// ====================================================================
size_t vector_push(vector* vect, TYPE val) {
  if ((vect != NULL) && ensure_capacity(vect)) {
    vect->content[vect->size] = val;
    ++(vect->size);
    return vect->size;
  }
  return 0;
}

// ====================================================================
int vector_set(vector* vect, size_t pos, TYPE val) {
  if (vect != NULL) {
    if (pos >= vect->size) vect->size = pos+1;
    if (ensure_capacity(vect)) {
      vect->content[pos] = val;
      return 1;
    }
  }
  return 0;
}

// ====================================================================
TYPE vector_get(vector const * vect, size_t pos) {
  TYPE result = (TYPE) 0;
  if (vect != NULL) {
    if (pos < vect->size) {
      result = vect->content[pos];
    }
  }
  return result;
}

// ====================================================================
void print_vector(vector const * v, size_t line_length) {
  printf("size: %zu\n", v->size);
  printf("allocated: %zu\n", v->allocated);
  puts("elements:");
  if (v->size > 0) {
    for (size_t i = 0, j = 1; i < v->size; ++i, ++j) {
      printf("%d ", vector_get(v, i));
      if (j >= line_length) {
        putchar('\n');
        j = 0;
      }
    }
    putchar('\n');
  } else {
    puts("<aucun>");
  }
}

// ======================================================================
int main(void)
{
  vector v;
  construct_vector(&v);

  vector_push(&v, 2);
  vector_set(&v, 3, -12);
  print_vector(&v, 10);
  putchar('\n');

  empty_vector(&v);
  print_vector(&v, 10);
  putchar('\n');

  vector_set(&v, 3 * VECTOR_PADDING + 5, -42);
  print_vector(&v, VECTOR_PADDING);

  destruct_vector(&v);
  return 0;
}
```

---

## Exercise 3: Matrix multiplications revisited (level 2)

### Part 1: first improvement: exercise on pointers

(file matrices2.c)

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

#define N 10

typedef struct {
   double m[N][N];
   size_t lignes;
   size_t colonnes;
} Matrice;

Matrice* lire_matrice(void);
void affiche_matrice(Matrice const *);
Matrice* multiplication(Matrice const * a, Matrice const * b);

/* ---------------------------------------------------------------------- */
int main(void)
{
  Matrice* M1 = NULL;
  Matrice* M2 = NULL;
  Matrice* M  = NULL;

  M1 = lire_matrice();
  if (M1 != NULL) {
    M2 = lire_matrice();
    if (M2 != NULL) {
      if (M1->colonnes != M2->lignes) {
        printf("Multiplication de matrices impossible !\n");
      } else {
        printf("Résultat :\n");
        M = multiplication(M1, M2);
        if (M != NULL) {
          affiche_matrice(M);
          free(M); M = NULL;
        }
      }
      free(M2); M2 = NULL;
    }
    free(M1); M1 = NULL;
  }
  return 0;
}

/* ---------------------------------------------------------------------- */
Matrice* lire_matrice(void)
{
  Matrice* resultat = NULL;

  /* On alloue la place mémoire pour la matrice de résultat */
  resultat = malloc(sizeof(Matrice));

  if (resultat != NULL) {
    size_t lignes;
    size_t colonnes;

    printf("Saisie d'une matrice :\n");

    do {
      printf("  Nombre de lignes (< %d) : ", N+1);
      scanf("%zu", &lignes);
    } while ((lignes < 1) || (lignes > N));

    do {
      printf("  Nombre de colonnes (< %d) : ", N+1);
      scanf("%zu", &colonnes);
    } while ((colonnes < 1)  || (colonnes > N));

    resultat->lignes = lignes;
    resultat->colonnes = colonnes;
    { size_t i, j;
    for (i = 0; i < lignes; ++i)
      for (j = 0; j < colonnes; ++j) {
        printf("  M[%zu,%zu]=", i+1, j+1);
        scanf("%lf", &resultat->m[i][j]);
      }
    }
  }

  return resultat;
}

/* ---------------------------------------------------------------------- */
Matrice* multiplication(Matrice const * a, Matrice const * b)
{
  Matrice* resultat = NULL;

  /* On alloue la place mémoire pour la matrice de résultat */
  resultat = malloc(sizeof(Matrice));
  if (resultat != NULL) {
    size_t i, j, k;

    resultat->lignes   = a->lignes;
    resultat->colonnes = b->colonnes;

    if (a->colonnes == b->lignes) {
      for (i = 0; i < a->lignes; ++i)
        for (j = 0; j < b->colonnes; ++j) {
          resultat->m[i][j] = 0.0;
          for (k = 0; k < b->lignes; ++k)
            resultat->m[i][j] += a->m[i][k] * b->m[k][j];
        }
    }
    else {
     resultat = NULL;
    }
  }
  return resultat;
}

/* ---------------------------------------------------------------------- */
void affiche_matrice(Matrice const * matrice)
{
  size_t i, j;
  for (i = 0; i < matrice->lignes; ++i) {
    for (j = 0; j < matrice->colonnes; ++j) {
      printf("%g ", matrice->m[i][j]);
    }
    putchar('\n');
  }
}
```

**Part 2 (level 3, optional): second improvement**

(file matrices3.c)

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

#define N 10

typedef struct {
   double m[N][N];
   size_t lignes;
   size_t colonnes;
} Matrice;

Matrice* lire_matrice(Matrice*);
void affiche_matrice(Matrice const *);
Matrice* multiplication(Matrice const * a, Matrice const * b,
                       /* pas de const ici, la valeur pointée par resultat *
                        * sera modifiée.                                   */
                        Matrice* resultat);

/* ---------------------------------------------------------------------- */
int main(void)
{
  Matrice M1, M2, M3;

  lire_matrice(&M1);
  if (multiplication(&M1, lire_matrice(&M2), &M3) == NULL) {
    printf("Multiplication de matrices impossible !\n");
  } else {
    printf("Résultat :\n");
    affiche_matrice(&M3);
  }
  return 0;
}

/* ---------------------------------------------------------------------- */
Matrice* lire_matrice(Matrice* resultat)
{
  if (resultat != NULL) {
    size_t lignes;
    size_t colonnes;

    printf("Saisie d'une matrice :\n");

    do {
      printf("  Nombre de lignes (< %d) : ", N+1);
      scanf("%zu", &lignes);
    } while ((lignes < 1) || (lignes > N));

    do {
      printf("  Nombre de colonnes (< %d) : ", N+1);
      scanf("%zu", &colonnes);
    } while ((colonnes < 1)  || (colonnes > N));

    resultat->lignes = lignes;
    resultat->colonnes = colonnes;
    { size_t i, j;
    for (i = 0; i < lignes; ++i)
      for (j = 0; j < colonnes; ++j) {
        printf("  M[%zu,%zu]=", i+1, j+1);
        scanf("%lf", &resultat->m[i][j]);
      }
    }
  }

  return resultat;
}

/* ---------------------------------------------------------------------- */
Matrice* multiplication(Matrice const * a, Matrice const * b,
                        Matrice* resultat)
{
  if (resultat != NULL) {
    size_t i, j, k;

    resultat->lignes   = a->lignes;
    resultat->colonnes = b->colonnes;

    if (a->colonnes == b->lignes) {
      for (i = 0; i < a->lignes; ++i)
        for (j = 0; j < b->colonnes; ++j) {
          resultat->m[i][j] = 0.0;
          for (k = 0; k < b->lignes; ++k)
            resultat->m[i][j] += a->m[i][k] * b->m[k][j];
        }
    } else {
      resultat = NULL;
    }
  }
  return resultat;
}
```

```
/* ---------------------------------------------------------------------- */
void affiche_matrice(Matrice const * matrice)
{
  size_t i, j;
  for (i = 0; i < matrice->lignes; ++i) {
    for (j = 0; j < matrice->colonnes; ++j) {
      printf("%g ", matrice->m[i][j]);
    }
    putchar('\n');
  }
}
```

### Part 3 (level 2): third improvement

(file matrices4.c)

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

#ifndef SIZE_MAX
#define SIZE_MAX (~(size_t)0)
#endif

typedef struct {
   double* m;
  /* Attention ici : on stocke le tableau en continu donc PAS DE double**. *
   * Ceux qui préfèrent double** auront une indirection de plus et un      *
   * tableau de pointeurs en plus en mémoire: perte de place !             *
   * Sans compter  que, comme ce sera présenté dans le cours 9, de telles  *
   * données ne seraiennt pas continues en mémoire.                        */

   size_t lignes;
   size_t colonnes;
} Matrice;

Matrice* empty(Matrice*);
void libere(Matrice*);
Matrice* redimensionne(Matrice*, size_t lignes, size_t colonnes);
Matrice* lire_matrice(Matrice*);
void affiche_matrice(Matrice const *);
Matrice* multiplication(Matrice const * a, Matrice const * b,
                        Matrice* resultat);

/* ---------------------------------------------------------------------- */
int main(void)
{
  Matrice M1, M2, M3;

  (void) lire_matrice(&M1);
  /* On met cet appel à lire_matrice ici et non pas dans l'appel de    *
   * multiplication() car on ne peut garantir l'ordre d'évaluation des *
   * arguments de l'appel (à multiplication)) et donc on ne peut       *
   * garantir que la lecture de M1 sera faite avant celle de M2.       *
   * Mettre cet appel ici permet de le garantir.                       */

  if (multiplication(&M1, lire_matrice(&M2), empty(&M3))
      /* Attention à ne pas oublier d'initialiser M3 !! */
      == NULL) {
    printf("Multiplication de matrices impossible !\n");
  } else {
    printf("Résultat :\n");
    affiche_matrice(&M3);
  }
```

```c
  libere(&M1);
  libere(&M2);
  libere(&M3);
  return 0;
}

/* ------------------------------------------------------------------------ */
Matrice* empty(Matrice* resultat)
{
  if (resultat != NULL) {
    resultat->lignes   = 0    ;
    resultat->colonnes = 0    ;
    resultat->m        = NULL ;
  }
  return resultat;
}

/* ------------------------------------------------------------------------ */
void libere(Matrice* resultat)
{
  if (resultat != NULL) {
    if (resultat->m != NULL) free(resultat->m);
    (void) empty(resultat);
  }
}

/* ------------------------------------------------------------------------ */
Matrice* lire_matrice(Matrice* resultat)
{
  if (resultat != NULL) {
    size_t lignes;
    size_t colonnes;

    do {
      printf("Saisie d'une matrice :\n");

      do {
        printf("  Nombre de lignes : ");
        scanf("%zu", &lignes);
      } while (lignes < 1);

      do {
        printf("  Nombre de colonnes : ");
        scanf("%zu", &colonnes);
      } while (colonnes < 1);

      resultat->lignes   = lignes;
      resultat->colonnes = colonnes;

      if (SIZE_MAX / lignes < colonnes) {
        resultat->m = NULL;
      } else {
        resultat->m = calloc(lignes*colonnes, sizeof(*(resultat->m)));
      }
      if (NULL == resultat->m) {
        printf("Matrice trop grande pour être allouée :-(\n");
      }

    } while (NULL == resultat->m);

    { size_t i, j;
      for (i = 0; i < lignes; ++i)
        for (j = 0; j < colonnes; ++j) {
          printf("  M[%zu,%zu]=", i+1, j+1);
          scanf("%lf", &resultat->m[i*resultat->colonnes+j]);
        }
    }
  }
  return resultat;
}

/* ---------------------------------------------------------------------- */
Matrice* redimensionne(Matrice* resultat, size_t lignes, size_t colonnes)
{
  if (resultat != NULL) {
    if (SIZE_MAX / lignes < colonnes) return NULL;
    if (resultat->lignes*resultat->colonnes < lignes*colonnes) {
      if ((lignes*colonnes) > SIZE_MAX / sizeof(*(resultat->m))) return NULL;
      double* const tmp = realloc(resultat->m, lignes*colonnes*sizeof(*(resultat->m)));
      if (NULL == tmp) {
        // don't change anything in case of failure
        return NULL;
      } else {
        // success => update
        resultat->m = tmp;
        resultat->lignes   = lignes;
        resultat->colonnes = colonnes;
      }
    }
  }
  return resultat;
}

/* ---------------------------------------------------------------------- */
Matrice* multiplication(Matrice const * a, Matrice const * b,
                        Matrice* resultat)
{
  if (resultat != NULL) {
    size_t i, j, k;

    if ((a->colonnes == b->lignes)
        && (redimensionne(resultat, a->lignes, b->colonnes) != NULL)) {
      for (i = 0; i < a->lignes; ++i)
        for (j = 0; j < b->colonnes; ++j) {
          resultat->m[i*resultat->colonnes+j] = 0.0;
          for (k = 0; k < b->lignes; ++k)
            resultat->m[i*resultat->colonnes+j] += a->m[i*a->colonnes+k]
                                                 * b->m[k*b->colonnes+j];
        }
    } else {
      resultat = NULL;
    }
  }
  return resultat;
}

/* ---------------------------------------------------------------------- */
void affiche_matrice(Matrice const * matrice)
{
  size_t i, j;
  const size_t imax = matrice->lignes*matrice->colonnes;
  for (i = 0; i < imax; i += matrice->colonnes) {
    for (j = 0; j < matrice->colonnes; ++j) {
      printf("%g ", matrice->m[i+j]);
    }
    putchar('\n');
  }
}
```

---

### Exercise 4: IP network

(file reseau.c)

```
// C99

#include <stdio.h>  // pour les entrées/sorties
#include <stdlib.h> // pour les allocations mémoire

#ifndef SIZE_MAX
#define SIZE_MAX (~(size_t)0)
#endif

/* ---------------------------------------------------------------------- *
 * Types de données                                                       *
 * ---------------------------------------------------------------------- */

typedef unsigned char IP_Addr[4]; // ou uint32_t de <stdint.h>

typedef struct _node {
  IP_Addr adresse;
  const struct _node** voisins; // Attention aux DEUX étoiles ici !
  // const optionnel (mais on ne modifie pas ses voisins ;-) )
  size_t nb_voisins;
  /*
    Note : on pourrait aussi ajouter un nb_allocated_voisins et faire
    de l'allocation de voisins page par page (au lieu de 1 par 1).
  */
} Noeud;

/* ---------------------------------------------------------------------- *
 * Prototypes (optionnel)                                                 *
 * ---------------------------------------------------------------------- */

Noeud* creation(const unsigned char adr1,
                const unsigned char adr2,
                const unsigned char adr3,
                const unsigned char adr4);

void sont_voisins(Noeud* p1, Noeud* p2);
// Pointeurs car les deux vont être modifiés (ajout de voisins).
// Autre type de retour possible (e.g. code d'erreur).

int ajoute_voisin(Noeud* p1, const Noeud* p2);
// Pensez MODULAIRE !
// const pointeur pour le second, non modifié ici.
// Retour : code d'erreur (optionnel, non utilisé d'ailleurs !)

unsigned int voisins_communs(const Noeud* p1, const Noeud* p2);
// const pointeurs pour éviter des copies inutiles.
// int ou size_t sont aussi valables commes type de retour.

void affiche(const Noeud* p);

void affiche_simple(const Noeud* p);
// Pensez MODULAIRE !

void libere(Noeud* p);
// NE PAS l'oublier !!
```

```c
/* ---------------------------------------------------------------------- */
int main()
{
  Noeud* rezo[] = {
    creation(192, 168,  1, 1),
    creation(192, 168,  1, 2),
    creation(192, 168,  1, 3),
    creation(192, 168, 10, 1),
    creation(192, 168, 10, 2),
    creation(192, 168, 20, 1),
    creation(192, 168, 20, 2)
  };

  for (size_t i = 0 ; i < sizeof(rezo) / sizeof(rezo[0]); ++i) {
    if (NULL == rezo[i]) {
      fprintf(stderr, "pas assez de mémoire\n");
      exit(-1);
    }
  }

  sont_voisins(rezo[0], rezo[1]);
  sont_voisins(rezo[0], rezo[2]);

  sont_voisins(rezo[1], rezo[2]);
  sont_voisins(rezo[1], rezo[3]);
  sont_voisins(rezo[1], rezo[5]);

  sont_voisins(rezo[2], rezo[3]);
  sont_voisins(rezo[2], rezo[5]);

  sont_voisins(rezo[3], rezo[4]);
  sont_voisins(rezo[3], rezo[5]);

  sont_voisins(rezo[5], rezo[6]);

  affiche(rezo[3]);

  affiche_simple(rezo[0]);
  printf(" et ");
  affiche_simple(rezo[5]);
  printf(" ont %u voisins communs.\n", voisins_communs(rezo[0], rezo[5]));

  affiche_simple(rezo[1]);
  printf(" et ");
  affiche_simple(rezo[2]);
  printf(" ont %u voisins communs.\n", voisins_communs(rezo[1], rezo[2]));

  /* garbage collecting */
  for (size_t i = 0 ; i < sizeof(rezo) / sizeof(rezo[0]); ++i) {
    libere(rezo[i]);
  }
  return 0;
}

/* ---------------------------------------------------------------------- *
 * Définitions                                                            *
 * ---------------------------------------------------------------------- */

// ======================================================================
Noeud* creation(const unsigned char adr1,
                const unsigned char adr2,
                const unsigned char adr3,
                const unsigned char adr4)
{
  Noeud* bebe = malloc(sizeof(Noeud));
  if (NULL == bebe) {
    fprintf(stderr, "Erreur (creation) : impossible d'allouer de la mémoire "
            "pour un nouveau Noeud (%u.%u.%u.%u).\n", adr1, adr2, adr3, adr4);
    return NULL;
  }

  bebe->adresse[0] = adr1;
  bebe->adresse[1] = adr2;
  bebe->adresse[2] = adr3;
  bebe->adresse[3] = adr4;

  bebe->voisins = NULL;
  bebe->nb_voisins = 0;

  return bebe;
}

// ======================================================================
int ajoute_voisin(Noeud* p1, const Noeud* p2)
{
  if (p1 != NULL) {
    if (NULL == p2) {
      fprintf(stderr, "Erreur (ajoute_voisin) : impossible d'ajouter un NULL-voisin\n");
      return 1;
    }

    ++(p1->nb_voisins);
    Noeud const ** const old_content = p1->voisins; // save, in case of error
    if ((p1->nb_voisins > SIZE_MAX / sizeof(Noeud*)) ||
        // NE PAS oublier de tester l'overflow !

        ((p1->voisins = realloc(p1->voisins, p1->nb_voisins * sizeof(Noeud*))) == NULL)
       ) {
      // echec
      p1->voisins = old_content;
      --(p1->nb_voisins);
      fprintf(stderr, "Erreur (ajoute_voisin) : %u.%u.%u.%u a déjà trop de voisins.\n",
              p1->adresse[0] , p1->adresse[1], p1->adresse[2], p1->adresse[3]);
      return 2;
    }

    p1->voisins[p1->nb_voisins-1] = p2;
    return 0;
  }
  return 3;
}

// ======================================================================
void sont_voisins(Noeud* p1, Noeud* p2)
{
  if (0 == ajoute_voisin(p1, p2)) {
    (void)ajoute_voisin(p2, p1);
  }
}

// ======================================================================
unsigned int voisins_communs(const Noeud* p1, const Noeud* p2)
{
  unsigned int voisins_commun = 0;
  if ((p1 != NULL) && (p2 != NULL)) {
    for (size_t i = 0; i < p1->nb_voisins; ++i) {
      for (size_t j = 0; j < p2->nb_voisins; ++j) {
        if (p1->voisins[i] == p2->voisins[j]) {
          ++voisins_commun;
        }
      }
    }
  }

  return voisins_commun;
}

// ======================================================================
void affiche(const Noeud* p)
{
  affiche_simple(p);
  printf(" a %zu voisins", p->nb_voisins);
  if (p->nb_voisins >= 1) {
    printf(" : ");
    if (p->nb_voisins >= 2) {
      for (size_t i = 0; i < p->nb_voisins - 1; ++i) {
        affiche_simple(p->voisins[i]);
        printf(", ");
      }
    }
    affiche_simple(p->voisins[p->nb_voisins - 1]);
    printf(".");
  }
  putchar('\n');
}

// ======================================================================
void affiche_simple(const Noeud* p)
{
  if (p != NULL) {
    printf("%u.%u.%u.%u"
          , p->adresse[0]
          , p->adresse[1]
          , p->adresse[2]
          , p->adresse[3]
          );
  } else {
    puts("(affiche_simple :) NULL");
  }
}

// ======================================================================
void libere(Noeud* p)
{
  free(p->voisins);
  free(p);
}
```

---

## Exercise 5 : Snake Game

(file snake-sol.c)

```
/* ====================================================================
 *
 * Exercice snake.c du cours
 * Programmation Orientee Systeme de M. Chappelier (Sections IN et SC).
 *
 * Si vous avez ncurses (libcurses-dev), compilez avec
 *  -DUSE_CURSES et -lncurses ; par exemple :
 * gcc -ansi -pedantic -Wall -DUSE_CURSES snake.c -o snake -lncurses
 *
 * ====================================================================
 */
#include <stdio.h>
#include <stdlib.h>

#ifdef USE_CURSES
#include <curses.h>
#define printf printw
#else
#define printw printf
#endif

/***************************************************************
 * Here come the data definitions
 ***************************************************************/
typedef struct {
    int dx;
    int dy;
} direction_t;

typedef enum {
    EMPTY, WALL, FOOD, SNAKE
} map_cell_t;

typedef struct snake_segment_t_ {
    unsigned int x;
    unsigned int y;
    int size;
    direction_t direction;
    struct snake_segment_t_ *prev;
} snake_segment_t;

typedef struct {
    snake_segment_t* head;
    snake_segment_t* tail;
} snake_t;

typedef struct {
    snake_t snake;
    unsigned int width;
    unsigned int height;
    map_cell_t* map;
} game_t;


/****************************************************************
 * This is the given game map.
 ****************************************************************/
#define MAP_WIDTH  80
#define MAP_HEIGHT 25

static map_cell_t const header_data[] = {
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,FOOD,
    FOOD,FOOD,FOOD,FOOD,FOOD,FOOD,FOOD,FOOD,FOOD,FOOD,FOOD,FOOD,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,FOOD,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,FOOD,FOOD,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,FOOD,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,FOOD,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,FOOD,EMPTY,
    EMPTY,EMPTY,FOOD,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,FOOD,EMPTY,
    EMPTY,EMPTY,EMPTY,FOOD,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,FOOD,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,FOOD,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,FOOD,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,FOOD,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
```

```c
    FOOD,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,FOOD,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,FOOD,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,FOOD,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,FOOD,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,FOOD,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,FOOD,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,
    EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,EMPTY,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,
    WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL,WALL
};

/*****************************************************************
 * Here come the function definitions
 *****************************************************************/
void snake_info(snake_t const* snake) {
    snake_segment_t const* seg;
    for (seg = snake->tail; seg; seg = seg->prev) {
        printf("(%02d,%02d) %d /%-d:%-d\n", seg->x, seg->y, seg->size,
               seg->direction.dx, seg->direction.dy);
    }
}

void snake_erase_tail(snake_t* snake)
{
  if (snake->tail != NULL) {
    snake_segment_t*  const newtail = snake->tail->prev;
    free(snake->tail);
    snake->tail = newtail;
  }
}

void snake_destroy(snake_t* snake)
{
  while (snake->tail != NULL) snake_erase_tail(snake);
}

int snake_add_segment(snake_t* snake, direction_t dir)
{
    snake_segment_t*  const seg = malloc(sizeof(snake_segment_t));
    if (!seg) {
        return -2;
    }

    seg->direction = dir;
    seg->prev = NULL;

    if (!snake->head) {
        snake->tail = seg;
        seg->size = 1;
    } else {
        seg->size = 0;
        seg->x = snake->head->x + dir.dx;
        seg->y = snake->head->y + dir.dy;
        snake->head->prev = seg;
    }
    snake->head = seg;

    return 0;
}

int snake_move(snake_t* snake, direction_t direction)
{
    if (snake->head->direction.dx == direction.dx &&
        snake->head->direction.dy == direction.dy) {
        snake->head->x += direction.dx;
        snake->head->y += direction.dy;
    } else {
        if (snake_add_segment(snake, direction) != 0) {
            return -1;
        }
    }

    if (snake->head == snake->tail) {
        return 0;
    }

    ++snake->head->size;
    --snake->tail->size;
    if (snake->tail->size == 0) {
      snake_erase_tail(snake);
    }

    return 0;
}

map_cell_t* cell(game_t* game, unsigned int x, unsigned int y) {
  return &(game->map[y*  game->width + x]);
}

int game_update(game_t* game, direction_t direction)
{
    snake_t* const snake = &game->snake;

    unsigned int const tail_x = snake->tail->x - (snake->tail->size-1) * snake->tail->direction.dx;
    unsigned int const tail_y = snake->tail->y - (snake->tail->size-1) * snake->tail->direction.dy;

    if (snake_move(snake, direction) != 0) {
        return -1;
    }

    switch (*cell(game, snake->head->x, snake->head->y)) {
    case WALL:
    case SNAKE:
      return -1;

    case FOOD:
      ++snake->tail->size;
      break;

    default:
      *cell(game, tail_x, tail_y) = EMPTY;
      break;
    }

    *cell(game, snake->head->x, snake->head->y) = SNAKE;

    return 0;
}

int game_init_snake(game_t* game, unsigned int orig_x, unsigned int orig_y)
{
    direction_t dir = {0,0};
    game->snake.head = NULL;
    game->snake.tail = NULL;

    if (snake_add_segment(&game->snake, dir) != 0) {
        return -1;
    }

    game->snake.head->x = orig_x;
    game->snake.head->y = orig_y;

    *cell(game, game->snake.head->x, game->snake.head->y) = SNAKE;

    return 0;
}

/****************************************************************/
int game_init_map(game_t* game, const map_cell_t* map, unsigned int width, unsigned int height)
{
    unsigned int x, y;
```

```c
    game->width = width;
    game->height = height;
    game->map = calloc(game->width * game->height, sizeof(map_cell_t));

    if (game->map == NULL) {
        game->width = game->height = 0;
        return -1;
    }

    for (y = 0; y < height; ++y) {
        for (x = 0; x < width; ++x) {
          *cell(game, x, y) = map[y*width + x];
        }
    }

    return 0;
}

/*****************************************************************/
game_t* game_init(unsigned int orig_x, unsigned int orig_y)
{
    game_t* game = malloc(sizeof(game_t));
    if (game != NULL) {
      if (game_init_map(game, header_data, MAP_WIDTH, MAP_HEIGHT) != 0) {
        free(game); game = NULL;
      } else if (game_init_snake(game, orig_x, orig_y) != 0) {
        free(game->map);
        free(game); game = NULL;
      }
    }
    return game;
}

/*****************************************************************/
void game_destroy(game_t* game)
{
    snake_destroy(&game->snake);
    free(game->map);
    free(game); game = NULL;
}

/****************************************************************
 * The following handles I/O and is not part of the game engine
 *****************************************************************/

void game_print(game_t* game)
{
    unsigned int x, y;
#ifdef USE_CURSES
    const int color = has_colors();
    clear();
    if (color) {
      start_color();
      init_pair(WALL , COLOR_BLACK, COLOR_YELLOW);
      init_pair(SNAKE, COLOR_BLACK, COLOR_GREEN );
      init_pair(FOOD , COLOR_BLACK, COLOR_RED   );
    }
#endif
    printw("\n");
    for (y = 0; y < game->height; ++y) {
        for (x = 0; x < game->width; ++x) {
            switch(*cell(game, x, y)) {
                case EMPTY:
                    printw(" ");
                    break;

                case WALL:
#ifdef USE_CURSES
                    if (color) {
                        attron(COLOR_PAIR(WALL));
                        printw(" ");
                        attroff(COLOR_PAIR(WALL));
                    } else
#endif
                    printw("O");
                    break;

                case FOOD:
#ifdef USE_CURSES
                    if (color) {
                        attron(COLOR_PAIR(FOOD));
                        printw(" ");
                        attroff(COLOR_PAIR(FOOD));
                    } else
#endif
                    printw("F");
                    break;

                case SNAKE:
#ifdef USE_CURSES
                    if (color) {
                        attron(COLOR_PAIR(SNAKE));
                        printw(" ");
                        attroff(COLOR_PAIR(SNAKE));
                    } else
#endif
                    printw("S");
                    break;
                default:
                    printw("?");
                    break;
            }
        }
        printw("\n");
    }
    /* For debugging */
    snake_info(&game->snake);
#ifdef USE_CURSES
    refresh();
#else
    getchar();
#endif
}

/* Transforms a keypress to dx and dy coordinates */
void handle_key_press(int key, direction_t* dir)
{
#ifndef USE_CURSES
#define KEY_DOWN  's'
#define KEY_UP    'w'
#define KEY_LEFT  'a'
#define KEY_RIGHT 'd'
#endif

    switch (key) {
```

```
    case KEY_DOWN:
        dir->dx =  0;    dir->dy =  1;
        break;

    case KEY_UP:
        dir->dx =  0;    dir->dy = -1;
        break;

    case KEY_RIGHT:
        dir->dx =  1;    dir->dy =  0;
        break;

    case KEY_LEFT:
        dir->dx = -1;    dir->dy =  0;
        break;

    default:
        dir->dx =  0;    dir->dy =  0;
        break;
    }
}

void game_loop(game_t* game)
{
    direction_t dir;
#ifdef USE_CURSES
    /* The user must move the snake manually, it does not move by itself */
#define getkey getch()
#else
    /* Change this array to simulates moves.
     * An x is a step where no keys are pressed */
    char const keys[] = "xsxdxxxxxxxxxxxxxxxxxwxxdxxwxxdxsxsxsx";
    const char* key = keys;
#define getkey (*key)
#endif
    do {
        game_print(game);
        handle_key_press(getkey, &dir);
        if (dir.dx == 0 && dir.dy == 0) {
            dir = game->snake.head->direction;
        }
    } while ((game_update(game, dir) == 0)
#ifndef USE_CURSES
             && (*++key)
#endif
            );
    printw("Game over\n");
}

int main(void)
{
    game_t* game;

#ifdef USE_CURSES
    initscr();
    raw();
    noecho();
    keypad(stdscr, TRUE);
#endif

    game = game_init(3,3);
    if (game) {
        game_loop(game);
        game_destroy(game);
    }

#ifdef USE_CURSES
    endwin();
#endif
    return 0;
}
```