Prog. Or. Système - Correction série 03 : Tableaux et structures en C

Exercice 1 : produit scalaire
Version élémentaire du code :

(fichier src/scalaire.c)

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

double scalaire(const double u[], const double v[], size_t taille);

int main(void)
{
  size_t const N_MAX = 10;
  double v1[N_MAX];
  double v2[N_MAX];

  size_t n = 2;

  do {
    printf("Quelle taille pour vos vecteurs [1 à %zu] ?\n", N_MAX);
    scanf("%zu", &n);
  } while ((n < 1) || (n > N_MAX));

  printf("Saisie du premier vecteur :\n");
  for (size_t i = 0; i < n; ++i) {
    printf(" v1[%zu] = ", i);
    scanf("%lf", &v1[i]);
  }

  printf("Saisie du second vecteur :\n");
  for (size_t i = 0; i < n; ++i) {
    printf(" v2[%zu] = ", i);
    scanf("%lf", &v2[i]);
  }

  printf("le produit scalaire de v1 par v2 vaut %lf\n",
         scalaire(v1, v2, n));
  return 0;
}

double scalaire(const double u[], const double v[], size_t taille)
{
```

```
  double somme = 0.0;
  for (size_t i = 0; i < taille; ++i) {
    somme += u[i] * v[i];
  }

  return somme;
}
```

Exercice 2 : multiplication de matrices
(fichier src/matrices.c)

```
/* C89 */
#include <stdio.h>

#define N 10

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

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

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

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

/* ---------------------------------------------------------------------- */
Matrice lire_matrice(void)
{
  Matrice resultat;
  size_t lignes = 2;
```

```
  size_t colonnes = 2;

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

  do {
    printf("  Nombre de lignes (< %d) : ", N+1);
    scanf("%lu", &lignes); /* "%zu" en C99 ; c'est mieux. */
  } while ((lignes < 1) || (lignes > N));

  do {
    printf("  Nombre de colonnes (< %d) : ", N+1);
    scanf("%lu", &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[%lu, %lu]=", i+1, j+1);
      scanf("%lf", &resultat.m[i][j]);
    }
  }

  return resultat;
}

/* ---------------------------------------------------------------------- */
Matrice multiplication(const Matrice a, const Matrice b)
{
  Matrice resultat = a; /* Disons que par convention on retourne a si la
                         * multiplication ne peut pas se faire.
                         */
  size_t i, j, k; /* variables de boucle */

  if (a.colonnes == b.lignes) {
    resultat.lignes = a.lignes;
    resultat.colonnes = b.colonnes;

    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];
      }
  }
```

```
  return resultat;
}

/* ------------------------------------------------------------------- */
void affiche_matrice(const Matrice 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');
  }
}
```

Note

Le type `Matrice` est défini comme étant un tableau de taille `N x N` (avec `N = 10`). Cela implique que dès qu'on crée une `Matrice`, on alloue de la place mémoire pour 10 x 10 chiffres, même si l'utilisateur décide ensuite que sa taille sera inférieure. Toutefois, on ne peut pas faire autrement, puisque dans ce cas, on n'a pas le droit de déclarer un tel tableau sans en spécifier la taille.
Pour faire mieux, il faut attendre les pointeurs...

Exercice 3 : placements sans recouvrements
(fichier src/recouvrement.c)

```
#include <stdio.h>

#define DIM 10

int const true = 1;
int const false = 0;
char const VIDE = '.';
char const PLEIN = '#';

typedef char Grille[DIM][DIM];

/* ------------------------------------------------------------------- */
int remplitGrille(Grille grille, size_t ligne, size_t colonne,
                  char direction, size_t longueur)
{
  /* direction de l'objet sur x et y (vaut -1, 0, ou 1) */
  int dx, dy;

  /* les coordonnee de la case à modifier */
  size_t i = ligne;
  size_t j = colonne;
```

```
  /* la longueur modifiée */
  size_t l;

  /* est-ce possible de mettre tout l'élement ? */
  int possible = true;

  switch (direction) {
  case 'N': dx = -1; dy =  0; break;
  case 'S': dx =  1; dy =  0; break;
  case 'E': dx =  0; dy =  1; break;
  case 'O': dx =  0; dy = -1; break;
  }

  /* avant de modifier la grille il faut vérifier si c'est possible
   * de placer l'objet.
   */
  for (l = 0;
       (possible) && (i < DIM) && (j < DIM) && (l < longueur);
       ++l, i += dx, j += dy)
    if (grille[i][j] != VIDE) /* cette case est déjà occupée */
      possible = false;  /* on ne peut donc pas mettre l'objet voulu */

  /* Si l == longueur c'est que j'ai pu placer l'objet sur toute sa longueur.
   * Il se pourrait en effet que je sois sorti de la boucle ci-dessus parce que
   *  i >= DIM ou j >= DIM... ...ce qui n'a pas modifié possible jusqu'ici
   */
  possible = possible && (l == longueur);

  if (possible)
    /* on met effectivement l'objet, plus besoin de test ici */
    for (l = 0, i = ligne, j = colonne; l < longueur;
         ++l, i += dx, j += dy)
      grille[i][j] = PLEIN;

  return possible;
}

/* ---------------------------------------------------------------------- */
void initGrille(Grille grille)
{
  size_t i, j;
  for (i = 0; i < DIM; ++i)
    for (j = 0; j < DIM; ++j)
      grille[i][j] = VIDE;
}
```

```
/* ---------------------------------------------------------------------- */
void ajouterElements(Grille grille)
{
  int x = 1, y = 1;
  char dir = 'S';
  size_t l = 1;

  do {
    printf("Entrez coord. x : ");
    scanf("%d", &x);

    if (x >= 0) {
      printf("Entrez coord. y : ");
      scanf("%d", &y);

      if (y >= 0) {
        getchar(); /* absorbe le \n qui traine */
        do {
          printf("Entrez direction (N,S,E,O) : ");
          scanf("%c", &dir);
        } while ((dir != 'N') && (dir != 'S') && (dir != 'E') && (dir != 'O'));

        printf("Entrez taille : ");
        scanf("%lu", &l);

        printf("Placement en (%d,%d) direction %c longueur %lu -> ",
               x, y, dir, l);

        if (remplitGrille(grille, x, y, dir, l))
          printf("succès");
        else
          printf("ECHEC");
        putchar('\n');

      }
    }
  } while ((x >= 0) && (y >= 0));
}

/* ---------------------------------------------------------------------- */
void afficheGrille(const Grille grille)
{
  size_t i, j;
  for (i = 0; i < DIM; ++i) {
    for (j = 0; j < DIM; ++j)
      printf("%c", grille[i][j]);
    putchar('\n');
```

```
  }
}

/* ---------------------------------------------------------------------- */
int main(void)
{
  Grille grille;

  initGrille(grille);
  ajouterElements(grille);
  afficheGrille(grille);
  return 0;
}
```

Exercice 4 : Nombres complexes (niveau 1)
(fichier src/complexe.c)

```
#include <stdio.h>

typedef struct {
  double x;
  double y;
} Complexe;

/* Solution simple */
void affiche(const Complexe z)
{
  printf("(%g,%g)", z.x, z.y);

  /* autre solution :  printf("%g+%gi", z.x, z.y); */
}

/* Solution plus complexe mais plus élégante */
void affiche2(const Complexe z)
{
  double y_affiche = z.y;

  if ((z.x == 0.0) && (z.y == 0.0)) {
    printf("0");
    return;
  }

  if (z.x != 0.0) {
    printf("%g", z.x);
    if (z.y > 0.0)
      printf("+"); /* ou putchar('+'); */
```

```
    else if (z.y < 0.0) {
      putchar('-');
      y_affiche = -z.y;
    }
  }
  if (y_affiche != 0.0) {
    if ((z.x == 0.0) && (y_affiche == -1.0))
      putchar('-');
    else if (y_affiche != 1.0)
      printf("%g", y_affiche);
    putchar('i');
  }
}

/* ------------------------------------ */
Complexe addition(const Complexe z1, const Complexe z2)
{
  Complexe z;
  z.x = z1.x + z2.x;
  z.y = z1.y + z2.y;
  return z;
}

/* ------------------------------------ */
Complexe soustraction(const Complexe z1, const Complexe z2)
{
  Complexe z;
  z.x = z1.x - z2.x;
  z.y = z1.y - z2.y;
  return z;
}

/* ------------------------------------ */
Complexe multiplication(const Complexe z1, const Complexe z2)
{
  Complexe z;
  z.x = z1.x * z2.x - z1.y * z2.y;
  z.y = z1.x * z2.y + z1.y * z2.x;
  return z;
}

/* ------------------------------------ */
Complexe division(const Complexe z1, const Complexe z2)
{
  const double r = z2.x*z2.x + z2.y*z2.y;
  Complexe z;
  z.x = (z1.x * z2.x + z1.y * z2.y) / r;
```

```
  z.y = (z1.y * z2.x - z1.x * z2.y) / r;
  return z;
}

/* ------------------------------------ */
int main(void)
{
  Complexe un = { 1.0, 0.0 };
  Complexe i  = { 0.0, 1.0 };
  Complexe j;
  Complexe z;
  Complexe z2;

  j = addition(un, i);

  affiche(un); printf(" + "); affiche(i); printf(" = ");
  affiche(j); putchar('\n');

  z = multiplication(i,i);
  affiche(i); printf(" * "); affiche(i); printf(" = ");
  affiche(z); putchar('\n');

  z = multiplication(j,j);
  affiche(j); printf(" * "); affiche(j); printf(" = ");
  affiche(z); putchar('\n');

  z2 = division(z,i);
  affiche(z); printf(" / "); affiche(i); printf(" = ");
  affiche(z2); putchar('\n');

  z.x = 2.0; z.y = -3.0;
  z2 = division(z,j);
  affiche(z); printf(" / "); affiche(j); printf(" = ");
  affiche(z2); putchar('\n');

  return 0;
}
```

Exercice 5 : Nombres complexes revisités (niveau 2)
(fichier src/complexe2.c)

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

// ------------------------------------------------------------------------
```

```
typedef struct {
  double x;
  double y;
} Complexe;

typedef struct {
  Complexe z1;
  Complexe z2;
} Solutions;

// ------------------------------------------------------------------------
void affiche(const Complexe z);

Complexe addition      (const Complexe z1, const Complexe z2);
Complexe soustraction  (const Complexe z1, const Complexe z2);
Complexe multiplication(const Complexe z1, const Complexe z2);
Complexe division      (const Complexe z1, const Complexe z2);
Complexe racine        (const Complexe z);

Solutions resoudre_second_degre(const Complexe b, const Complexe c);

// ========================================================================
int main(void)
{
  Complexe b  = { 3.0, -2.0 };
  Complexe c  = { -5.0, 1.0 };

  Solutions s = resoudre_second_degre(b, c);

  printf("Avec b="); affiche(b);
  printf(" et c=");  affiche(c);
  printf(" on a :\n");
  printf("  z1="); affiche(s.z1); putchar('\n');
  printf("  z2="); affiche(s.z2); putchar('\n');

  return 0;
}

// ========================================================================
void affiche(const Complexe z)
{
  if ((z.x == 0.0) && (z.y == 0.0)) {
    printf("0");
    return;
  }

  double y_affiche = z.y;
```

```
  if (z.x != 0.0) {
    printf("%g", z.x);
    if (z.y > 0.0)
      putchar('+');
    else if (z.y < 0.0) {
      putchar('-');
      y_affiche = -z.y;
    }
  }
  if (y_affiche != 0.0) {
    if ((z.x == 0.0) && (y_affiche == -1.0))
      putchar('-');
    else if (y_affiche != 1.0)
      printf("%g", y_affiche);
    putchar('i');
  }
}

// ============================================================================
Complexe addition(const Complexe z1, const Complexe z2)
{
  return (Complexe) { z1.x + z2.x, z1.y + z2.y };
}

// ============================================================================
Complexe soustraction(const Complexe z1, const Complexe z2)
{
  return (Complexe) { z1.x - z2.x, z1.y - z2.y };
}

// ============================================================================
Complexe multiplication(const Complexe z1, const Complexe z2)
{
  return (Complexe) {
    z1.x * z2.x - z1.y * z2.y ,
    z1.x * z2.y + z1.y * z2.x
  };
}

// ============================================================================
Complexe division(const Complexe z1, const Complexe z2)
{
  const double r = z2.x*z2.x + z2.y*z2.y;
  return (Complexe) {
    (z1.x * z2.x + z1.y * z2.y) / r ,
    (z1.y * z2.x - z1.x * z2.y) / r
```

```
  };
}

// ========================================================================
Complexe racine(const Complexe z)
{
  const double r = sqrt(z.x * z.x + z.y * z.y);
  Complexe retour;

  retour.x = sqrt((r + z.x) / 2.0);
  if (z.y >= 0.0)
    retour.y = sqrt((r - z.x) / 2.0);
  else
    retour.y = - sqrt((r - z.x) / 2.0);

  return retour;
}

// ========================================================================
Solutions resoudre_second_degre(const Complexe b, const Complexe c)
{
  // Pour faciliter l'écriture
  const Complexe deux   = { 2.0, 0.0 };
  const Complexe quatre = { 4.0, 0.0 };

  // delta^2 = b^2 - 4c
  const Complexe sdelta = racine(soustraction(multiplication(b, b),
                                              multiplication(quatre, c)));

  // Calcule -b (ou alors faire une fonction "oppose")
  const Complexe mb = { -b.x, -b.y };

  // Réponse = -b +- delta / 2
  return (Solutions) {
    division( soustraction(mb, sdelta) , deux) ,
    division( addition    (mb, sdelta) , deux)
  };
}
```

Dernière mise à jour : $Date: 2012-03-23 17:31:02 $ ($Revision: 1.2 $)