Prog. Or. Système - Correction série 02 : Variables et opérateurs en C

Exercice 1 : livret
(fichier src/livret.c)

```
// C99, pour changer un peu
#include <stdio.h>

int main(void)
{
  printf("     Tables de multiplication\n");

  // On itère sur les tables de 2 à 10 (compris)
  for (int i = 2; i <= 10; ++i) {
    printf("\n Table de %d :\n", i);

    // Pour chaque table, on itère de 1 à 10 (multiplicateurs)
    for (int j = 1; j <= 10; ++j) {

      // Affichage de la multiplication effectuée et son résultat
      printf("    %d * %d = %d\n", i, j, i*j);

    }
  }

  return 0;
}
```

Exercice 2 : rebonds de balles
(fichier src/rebond1.c)

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

/* On déclare g constante. Elle ne peut plus être modifiée dans le   *
 * reste du code                                                     */
double const g = 9.81;

int main(void)
{
    /* Déclarations */
```

```
double v = 0.0, v1 = 0.0;   /* vitesses avant et après le rebond */
double h = 0.0, h1 = 0.0;   /* hauteur avant le rebond, hauteur de remontée */
double H0 = 1.0, eps = 0.1; /* hauteur initiale, coefficient de rebond */
int nombre = 0, NBR = 1;

/*
 * Entrée des valeurs par l'utilisateur,
 * avec test de validité
 */

do {
    printf("Coefficient de rebond (0 <= coeff < 1) :\n");
    scanf("%lf", &eps);

/* répétition tant que l'utilisateur n'entre pas une valeur
 * correcte pour eps                                          */
} while ( (eps < 0.0) || (eps >= 1.0) );

do {
    printf("Hauteur initiale      (0 <= H0)        :\n");
    scanf("%lf", &H0);

/* répétition tant que l'utilisateur n'entre pas une valeur
 * positive pour H0                                           */
} while ( H0 < 0.0);

do {
    printf("Nombre de rebonds     (0 <= N )        :\n");
    scanf("%d", &NBR);

/* répétition tant que l'utilisateur n'entre pas une valeur
 * positive pour NBR                                          */
} while (NBR < 0);

/* ========== Boucle de calcul ========== */

/* Au départ (0 rebond), la hauteur de rebond vaut H0 */
h = H0;

for (nombre = 0; nombre < NBR; ++nombre) {
    v  = sqrt(2.0 * g * h);
    v1 = eps * v;              /* vitesse après le rebond */
    h1 =(v1 * v1) / (2 * g);   /* la hauteur à laquelle elle remonte... */
    h  = h1;                   /* ...qui devient la nouvelle hauteur initiale */

    printf("rebond %d : %f\n", nombre+1, h);
```

```
    }

    /* Affichage du résultat */
    printf("Au %dème rebond, la hauteur sera de %f m.\n", NBR, h);

    return 0;
}
```

Note:

dans la boucle `for` de ce code, la variable `nombre` est initialisée à `0`. Pourtant, lors du premier passage dans cette boucle, nous sommes déjà en train de calculer le *premier* (1) rebond. C'est pourquoi, à la ligne:

```
printf("rebond %d : %f
", nombre+1, h);
```

le nombre de rebonds affiché est `nombre+1` et pas `nombre`. Une solution pour éviter cette addition serait d'initialiser directement: `nombre = 1` dans la boucle, mais *attention !*, cela impliquerait une modification de la condition d'arrêt de la boucle également.

Exercice 3 : rebonds de balles (2)
(fichier src/rebond2.c)

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

double const g = 9.81;

int main(void)
{
    /* Déclarations */

    double v = 0.0, v1 = 0.0;               /* vitesses avant et après le rebond */
    double h = 0.0, h1 = 0.0;               /* hauteur avant le rebond, hauteur de
remontée */
    double H0 = 1.0, eps = 0.1, h_fin = 0.1; /* entrées de l'utilisateur */
    int nombre = 0;

    /*
     * Entrée des valeurs par l'utilisateur,
     * avec test de validité
     */

    do {
        printf("Coefficient de rebond (0 <= coeff < 1) :\n");
        scanf("%lf", &eps);
```

```
    } while ( (eps < 0.0) || (eps >= 1.0) );
    do {
        printf("Hauteur initiale      (0 <= H0)        :\n");
        scanf("%lf", &H0);
    } while ( H0 < 0.0 );

    do {
       printf("Hauteur finale  (0 <= h_fin ) :\n");
       scanf("%lf", &h_fin);
    } while ((h_fin < 0.0) || (h_fin > H0));

    /* Boucle de calcul */

    h = H0;
    nombre = 0;
    do {
        v  = sqrt(2.0 * g * h);
        v1 = eps * v;               /* vitesse après le rebond */
        ++nombre;                   /* incrémente le nombre de rebonds */
        h1 = (v1 * v1) / (2.0 * g); /* la hauteur à laquelle elle remonte... */
        h  = h1;                    /* ...qui devient la nouvelle hauteur initiale */

        /* nombre est incrémenté plus haut,
         * il peut donc être affiché directement ici
         * (par opposition à l'exercice précédent)
         */
        printf("rebond %d : %f\n", nombre, h);

    } while(h1 > h_fin);

    /* Affichage du résultat */

    printf("Nombre de rebonds : %d\n", nombre);
    return 0;
}
```

Exercice 4 : histoire de prêt
(fichier src/pret.c)

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

int main(void)
{
    double cumul = 0.0, S = 0.0, S0 = 0.0, r = 0.0, ir = 0.0;
    int nbr = 0;   /* nombre de remboursements */
```

```
    do {
        printf("Somme prêtée                    (S0 > 0) :\n");
        scanf("%lf", &S0);
    } while (S0 <= 0.0);

    do {
        printf("Montant fixe remboursé chaque mois (r > 0) :\n");
        scanf("%lf", &r);
    } while (r <= 0.0);

    do {
        printf("Taux d'intérêt en %% (0 < tx < 100) :\n");
        scanf("%lf", &ir);
    } while ( (ir <= 0) || (ir >= 100) );
    ir /= 100.00;  /* le taux d'intérêt a été donné en %, nous le transformons */

    S = S0;        /* initialisation de la somme résiduelle à la somme initiale */
    while (S > 0.0) {
        ++nbr;     /* on incrémente le nombre de mois requis */
        cumul = cumul + ir * S;
        S = S - r; /* on décrémente la somme résiduelle */
        printf("%d: S=%f, cumul=%f\n", nbr, S, cumul); /* pour info, non demandé */
    }

    /* la valeur %.2f représente un réel avec 2 chiffres après la virgule */
    printf("Somme des intérêts encaissés : %.2f (sur %d mois).\n", cumul, nbr);

    return 0;
}
```

Exercice 5 : nombres premiers
(fichier src/premiers.c)

```
// C99, pour changer
#include <stdio.h>
#include <math.h>

int main(void)
{
    // Saisie du nombre à tester
    int n = 2;
    do {
        printf("Entrez un nombre entier > 1 :\n");
        scanf("%d", &n);
    } while (n <= 1);
```

```
    int diviseur = 1;  // Diviseur trouvé. Si c'est 1, alors le nombre est premier

    if (0 == (n % 2)) {
        // Le nombre est pair
        if (n != 2) {
            diviseur = 2; // n n'est pas premier (2 est diviseur)
        }
    } else {
        const double borne_max = sqrt((double) n);
        for (int i = 3; (diviseur == 1) && (i <= borne_max); i += 2) {
            if (0 == (n % i)) {
                diviseur = i; // n n'est pas premier (i est diviseur)
            }
        }
    }

    printf("%d", n);

    if (diviseur == 1) {
        printf(" est premier");
    } else {
        printf(" n'est pas premier, car il est divisible par %d", diviseur);
    }
    printf("\n");
    return 0;
}
```

Quelques explications:

La valeur `n` entrée par l'utilisateur doit être un entier. Toutefois, si l'on veut calculer la racine carrée de ce nombre, il est nécessaire de le transformer momentanément en double. Cette opération s'effectue comme indiqué à la ligne :

```
const double borne_max = sqrt((double) n);
```

en spécifiant le type souhaité entre parenthèses avant la variable.

Dans la boucle `for` à la fin du programme, il y a une double condition :

1. on vérifie qu'on n'a pas trouvé un diviseur à l'itération précédente en testant la valeur de `premier`. On pourrait éviter ce test en insérant un `break` à la dernière ligne de la condition `if`, mais l'utilisation de `break` et `continue` est très fortement déconseillée.
2. on vérifie qu'on n'a pas encore dépassé la `borne_max`.

Résultats :

```
2 est premier
16 n'est pas premier, car il est divisible par 2
17 est premier
91 n'est pas premier, car il est divisible par 7
589 n'est pas premier, car il est divisible par 19
1001 n'est pas premier, car il est divisible par 7
1009 est premier
1299827 est premier
2146654199 n'est pas premier, car il est divisible par 46327
```

Remarques :

Si vous souhaitez tester des nombres plus grands que `2147483647` (c'est-à-dire 2^{31}-1, qui d'ailleurs est premier !), remplacez
`int n;`
par
`unsigned long n;`

Il faut aussi changer la déclaration de `i` :
`unsigned long i;`

Vous pouvez alors tester jusqu'à `4294967295` (Essayez par exemple `4292870399`).
La signification de tout ceci sera vue plus tard dans le cours.

Pour ceux qui aimeraient tester de plus grand nombre encore, vous pouvez remplacer les entiers précédents (`int` puis `unsigned long`) par :
`unsigned long long n;`
(n'oubliez pas de le faire pour `n` et pour `i`; et notez que ce type étendu n'est disponible que depuis C99]

Ceci vous permet d'aller jusqu'à `18446744073709551615` (Essayez par exemple `18446744073709551577` ou `18446744073709551557` (il faut attendre un petit moment).

Le format `printf` pour les `long unsigned` est `"%lu"` et pour les `long long unsigned`, `"%llu"`.

Exercice 6 : expressions arithmétiques
(fichier src/expressions.c)

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

int main(void)
```

```
{
  double x = 0.0;                       /* déclaration */
  double resultat = 0.0;
  printf("Entrez un nombre réel : "); /* message */
  scanf("%lf", &x);                    /* lecture de x */

  /* Expression 1 */
  printf("Expression 1 : ");
  if (0.0 == x) {
      printf("non définie\n");
  }
  else {
      resultat = x / (1.0 - exp(x));
      printf("%f\n", resultat);
  }

  /* Expression 2 */
  printf("Expression 2 : ");
  if ((x <= 0.0) || (1.0 == x)) {
      printf("non définie\n");
  }
  else {
      resultat = x * log(x) * exp( 2.0 / (x-1.0) ) ;
      printf("%f\n", resultat);
  }

  /* Expression 3 */
  printf("Expression 3 : ");
  if ( (x > 0.0) && (x < 8.0) ) {
      printf("non définie\n");
  }
  else {
      resultat = ( -x - sqrt(x*x - 8.0*x) ) / (2.0-x) ;
      printf("%f\n", resultat);
  }

  /* Expression 4 */
  printf("Expression 4 : ");
  if  ((x <= 1.0 ) && (x >= 0.0)) {
      printf("non définie\n");
  }
  else  {
      resultat = (sin(x) - x/20.0) * log(sqrt(x*x - 1.0/x));
      if (resultat <= 0.0) {
          printf("non définie\n");
      } else {
          printf("%f\n", sqrt(resultat));
```

```
      }
  }

  return 0;
}
```

Exemple de déroulement :

```
./formules
Entrez un nombre réel : -3.5
Expression 1 : -3.608982
Expression 2 : non définie
Expression 3 : -0.517143
Expression 4 : 0.815318

./formules
Entrez un nombre réel : -1
Expression 1 : -1.581977
Expression 2 : non définie
Expression 3 : -0.666667
Expression 4 : non définie

./formules
Entrez un nombre réel : 0
Expression 1 : non définie
Expression 2 : non définie
Expression 3 : -0.000000
Expression 4 : non définie

./formules
Entrez un nombre réel : 1
Expression 1 : -0.581977
Expression 2 : non définie
Expression 3 : non définie
Expression 4 : non définie

./formules
Entrez un nombre réel : 2
Expression 1 : -0.313035
Expression 2 : 10.243407
Expression 3 : non définie
Expression 4 : 0.711989

./formules
Entrez un nombre réel : 3
Expression 1 : -0.157187
```

```
Expression 2 : 8.959013
Expression 3 : non définie
Expression 4 : non définie

./formules
Entrez un nombre réel : 8
Expression 1 : -0.002685
Expression 2 : 22.137106
Expression 3 : 1.333333
Expression 4 : 1.106779
```

Exercice 7 : équation du 3e degré
(fichier src/deg3.c)

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

/* ligne pour avoir M_PI (= pi). A mettre AVANT le include de math.h. */
#define  _USE_MATH_DEFINES
#include <math.h>
/* Si vraiment le compilateur respecte strictement le standard, même *
 * _USE_MATH_DEFINES ne fera pas l'affaire. On fait alors le travail *
 * nous même :                                                       */
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

int main(void)
{
  double a0 = 0.0, a1 = 0.0, a2 = 0.0,
    z1 = 0.0, z2 = 0.0, z3 = 0.0,
    D = 0.0, Q = 0.0, R = 0.0, S = 0.0, T = 0.0;

  printf("Entrez a2, puis a1, puis a0 :\n");
  scanf("%lf %lf %lf", &a2, &a1, &a0);

  Q = (3.0 * a1 - a2*a2) / 9.0;
  R = (9.0 * a2 * a1 - 27.0 * a0 - 2.0 * a2*a2*a2) / 54.0;
  D = Q*Q*Q + R*R;
  printf("(pour info D = %f)\n", D);

  if (D < 0.0) { /* test du déterminant */
     /* cas de trois racines réelles */

     T  = acos( R / sqrt(-Q*Q*Q) );
     z1 = 2.0 * sqrt(-Q) * cos(T/3.0)              - a2/3.0;
     z2 = 2.0 * sqrt(-Q) * cos( (T+2*M_PI) / 3.0 ) - a2/3.0;
```

```
    z3 = 2.0 * sqrt(-Q) * cos( (T+4*M_PI) / 3.0 ) - a2/3.0;
    printf("Trois racines ( %f , %f , %f )\n", z1, z2, z3);
  } else {
    /* cas de moins de trois racines réelles */

    /* calcul de S */
    double s = R+sqrt(D);
    const double un_tiers = 1.0/3.0;
    if (0.0 == s)     { S = 0.0;                }
    else if (s < 0.0) { S = -pow(-s, un_tiers); }
    else if (s > 0.0) { S =  pow( s, un_tiers); }

    /* calcul de T */
    s = R-sqrt(D);
    if (0.0 == s)     { T = 0.0;                }
    else if (s < 0.0) { T = -pow(-s, un_tiers); }
    else if (s > 0.0) { T =  pow( s, un_tiers); }

    printf("(pour info S = %f, T = %f, S+T= %f)\n", S, T, S+T);

    /* calcul des solutions */
    z1 = -a2 / 3.0 + S + T;
    if ((0.0 == D) && (S+T != 0.0)) {
      z2 = -a2 / 3.0 - (S + T) / 2.0;
      printf("Deux racines...\n");
      printf("   l'une simple   : %f\n", z1);
      printf("   l'autre double : %f\n", z2);
    } else {
      printf("Une seule racine : %f\n", z1);
    }
  }

  return 0;
}
```

Remarque : la version proposée n'est pas « propre » numériquement. Vous savez certainement déjà que l'on ne devrait ***jamais*** faire de test d'égalité (`==`) avec des `double` (ça n'a pas de sens, numériquement : manque de précision sur les valeurs). Il vaudrait mieux définir une fonction (par exemple `double_eq` ou `double_cmp`) qui prennent trois `double`, les deux nombres à comparer et une précision, et retourne `0` si la valeur absolue de la différence entre les deux nombres est supérieure à la précision, et quelque chose de non nul sinon.

Je vous propose d'implémenter cette amélioration en guise d'exercice complémentaire.

Exercice 8 : Prototypes

Solution de 1,2,3 :

(fichier src/demander_nombre_1.c)

```
#include <stdio.h>

int demander_nombre(void);

int main(void)
{
    printf("Le nombre entré est : %d\n", demander_nombre());
    return 0;
}

int demander_nombre(void)
{
  int res = 0;
  printf("Entrez un nombre entier : ");
  scanf("%d", &res);
  return res;
}
```

Concernant le déplacement de la défintion de `demander_nombre` **après** l'appel (i.e. après le `main`) **sans** mettre de protoype avant, normalement le compilateur devrait «râler» et signaler lors de l'appel que la fonction `demander_nombre` n'est pas déclarée.

Cependant, C dans sa version de base (ANSI) est plus permissif et autorise ce que l'on appelle la définition *implicite* des fonctions (cette définition correspond à une fonction dont le nombre de paramètre est inconnu et retournant un entier, i.e. «~`int f();`~» [*sans* `void`]). [cf C90 3.3.2.2.Semantics]

Pour voir qu'il y a en effet un problème, ajoutez l'option `-Wall` (qui signifie «*all Warnings*») à la compilation :

```
gcc -Wall -o proto proto.c
```

Vous aurez cette fois bien un message d'alerte sur la définition implicite de la fonction `demander_nombre`.

Si vous utilisez le standard C99 (au lieu du standard ANSI), lequel **interdit** la définition implicite des fonctions [C99 6.5.2.2], vous devriez avoir une erreur :

```
gcc -std=c99 -o proto proto.c
```

(cependant toutes les versions de gcc ne suivent pas la norme sur ce point).

Solution de 4 :

(fichier src/demander_nombre_2.c)

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

```
int demander_nombre(int, int);

int main(void)
{
    printf("Le nombre entré est : %d\n", demander_nombre(1, 100));
    return 0;
}

int demander_nombre(int a, int b)
{
    int res = 0;

   /* échange a et b si ils ne sont pas dans le bon ordre.
    * Ceci est NÉCESSAIRE si on ne veut pas de boucle infinie
    * dans le cas ou on appelle la fonction avec a>b !!
    */
    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;
}
```

Note

Remarquez comme il est possible d'omettre le nom des paramètres dans un prototype: ici, le prototype `demander_nombre` ne spécifie que le type des paramètres, spéparés par des ",". Donner un nom aux paramètres n'est nécessaire que lors de la définition.

Solution de 5 :

(fichier src/demander_nombre.c)

```
#include <stdio.h>

int demander_nombre(int min, int max);

int main(void)
{
    printf("Le nombre entré est : %d\n", demander_nombre(1, 100));
    printf("Le nombre entré est : %d\n", demander_nombre(10, -1));
    return 0;
}
```

```
int demander_nombre(int a, int b)
{
    int res = 0;

    do {
      printf("Entrez un nombre entier ");
      if (a >= b)
        printf("supérieur ou égal à %d", a);
      else
        printf("compris entre %d et %d", a, b);
      printf(" :\n");
      scanf("%d", &res);
    } while ((res < a) || ((a < b) && (res > b)));

   return res;
}
```

Exercice 9 : Calcul approché d'une intégrale
(fichier src/integrale.c)

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

double f(double x) {
  return sin(x);
}

double demander_nombre(void)
{
  double res = 0;
  printf("Entrez un nombre réel : ");
  scanf("%lf", &res);
  return res;
}

double integre(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)
{
  double
    a = demander_nombre(),
    b = demander_nombre();
  printf("Integrale de sin(x) entre %f et %f : %.12f\n", a, b, integre(a, b));
  return 0;
}
```

Exercice 10 : La fonction cosinus
(fichier src/cos.c)

```
#include <stdio.h>

int demander_nombre(int a, int b)
{
    int res = 0;

   do {
       printf("Entrez le degré d'approximation entre %d et %d : ",
              a, b);
       scanf("%d", &res);
    } while ((res < a) || (res > b));
    return res;
}

double factorielle(int k)
{
   /* rappel: factorielle(0) = 1 et factorielle(1) = 1 */
   double fact = 1.0;
    int i;
    for(i = 2; i <= k; ++i) fact *= i;
    return fact;
}

double somme_partielle(double x, int N)
{
  double current_approx = 0.0; /* approximation courante            */
  double powerx = 1.0;       /* puissance de x (initialisée à x^0=1) */
    int i;
```

```
    for(i = 0; i < N; ++i)
    {
        if (i%2 == 0) {
            current_approx += powerx / factorielle(i*2);
        } else {
            current_approx -= powerx / factorielle(i*2);
        }
        powerx *= x*x; /* powerx is x^0, x^2, x^4, x^6, ...x^2n */
    }

    return current_approx;
}

int main(void)
{
    int N = demander_nombre(2, 170);
    double x = 0.0;

    do {
        printf("Entrez une valeur x pour le calcul de cos(x) : ");
        scanf("%lf", &x);
        printf("cos(x) ~ %.12f\n", somme_partielle(x,N));
    } while (x != 0.0);
    return 0;
}
```

Exercice 11 : histoires de dates

1. Le bug :

   quand `days` vaut 366 et `year` est une année bissextile quelconque (i.e. le dernier jour d'une année bissextile, comme par exemple le fameux 31 décembre 2008), le code reste dans le `while (days > 365)`, passe dans le premier `if (IsLeapYear(year))` mais ne satisfait pas `if (days > 366)`. on sort du premier if et recommence un tour dans la boucle (infinie).

2. Une solution :

   le problème vient donc du fait que l'on mélange la boucle sur le calcul du nombre d'années et le nombres de jours dans l'année (365 ou 366). Les solutions consistent à clairement séparer les deux. En voici une possible :

   (fichier src/zune.c)

```
/*
 * zune.c
 * ANSI C89
 */
```

```
#include <stdio.h>
#include <stdlib.h> /* atoi(3) */

#define MICROSOFT_EPOCH_YEAR 1980
#define december_31_2008 10593

typedef enum {
    JANUARY = 1,
    FEBRUARY,
    MARCH,
    APRIL,
    MAY,
    JUNE,
    JULY,
    AUGUST,
    SEPTEMBER,
    OCTOBER,
    NOVEMBER,
    DECEMBER
} Month;

int IsLeapYear(int y)
{

    return (((y % 4 == 0) && (y % 100 != 0)) || (y % 400 == 0));
}

/* on a besoin de l'année (year) pour le cas spécial (Février) */
int DaysForMonth(int year, Month month)
{
    int days = 31;

    switch (month) {
    case FEBRUARY: /* cas spécial */
        if (IsLeapYear(year))
            days = 29;
        else
            days = 28;
        break;
    case APRIL:
    case JUNE:
    case SEPTEMBER:
    case NOVEMBER:
        days = 30;
        break;
    default:
```

```
        days = 31;
    }

    return days;
}

int main(void)
{
    int year = MICROSOFT_EPOCH_YEAR;
    int days = 1, d = 31;
    Month month = JANUARY;

    printf("Entrez le nombre de jours écoulés depuis le 31/12/1979 : ");
    scanf("%d", &days);

    /* calcul de year */
    while (days > 0) {
        if (IsLeapYear(year))
            days -= 366;
        else
            days -= 365;
        ++year;
    }
    --year;
    if (IsLeapYear(year))
        days += 366;
    else
        days += 365;

    /* calcul de month */
    month = JANUARY;
    d = DaysForMonth(year, month);
    while (days > d) {
        days -= d;
        d = DaysForMonth(year, ++month);
    }

    printf("%02d/%02d/%d\n", days, month, year);

    return 0;
}
```

3. Sur la base de ce qui a été fait ci-dessus :
   (fichier src/unix-time.c)

```
/*
```

```
 * unix-time.c
 * ANSI C89
 */

#include <stdio.h>
#include <time.h>

#define UNIX_EPOCH_YEAR 1970

/* copié de zune.c */
typedef enum {
    JANUARY = 1,
    FEBRUARY,
    MARCH,
    APRIL,
    MAY,
    JUNE,
    JULY,
    AUGUST,
    SEPTEMBER,
    OCTOBER,
    NOVEMBER,
    DECEMBER
} Month;

int IsLeapYear(int y)
{

    return (((y % 4 == 0) && (y % 100 != 0)) || (y % 400 == 0));
}

/* on a besoin de l'année (year) pour le cas spécial (Février) */
int DaysForMonth(int year, Month month)
{
    int days = 31;

    switch (month) {
    case FEBRUARY: /* cas spécial */
        if (IsLeapYear(year))
            days = 29;
        else
            days = 28;
        break;
    case APRIL:
    case JUNE:
    case SEPTEMBER:
    case NOVEMBER:
```

```
        days = 30;
        break;
    default:
        days = 31;
    }

    return days;
}

int main(void)
{
    time_t now = time(NULL);
    time_t seconds, minutes, hours;
    int year = UNIX_EPOCH_YEAR;
    time_t days = 1, d = 31;
    Month month = JANUARY;

    printf("%u secondes se sont ecoulees depuis le 1.1.1970 a minuit.\n", now);

    seconds = now % 60;
    now /= 60;
    minutes = now % 60;
    now /= 60;
    hours = now % 24;
    now /= 24;

    /* copié de zune.c */
    /* calcul de year */
    days = now;
    year = UNIX_EPOCH_YEAR;
    while (days > 0) {
        if (IsLeapYear(year))
            days -= 366;
        else
            days -= 365;
        ++year;
    }
    --year;
    if (IsLeapYear(year))
        days += 366;
    else
        days += 365;

    /* calcul de month */
    month = JANUARY;
    d = DaysForMonth(year, month);
    while (days > d) {
```

```
        days -= d;
        d = DaysForMonth(year, ++month);
    }
    ++days; /* 0 corespond au 1er Janvier 1970, il faut faire +1 */

    printf("Nous sommes donc le %02d/%02d/%d a %02d:%02d:%02d\n",
            days, month, year, hours, minutes, seconds);

    return 0;
}
```

Oui, il y a un décalage d'une heure. La fonction `time(2)` renvoie le temps UTC (Coordinated Universal Time) alors qu'en Suisse le fuseau horaire est de GMT+1 (en hiver), ce qui explique que le programme `unix-time` affiche une heure en retard par rapport à l'heure suisse.

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