QUELQUES DOCUMENTS

Bernard PICHON
Laboratoire Cassiopée,
UMR 6202 CNRS
Observatoire de la Côte d'Azur à Nice
BP 4229 06304 NICE Cedex 04
Courriel : Bernard.Pichon@oca.eu

- Les problèmes de portabilité du C i.e. *int* et *char* (1 page)
- De la bonne utilisation des prototypes (2 pages)
- Comment 'bien' déclarer et utiliser en C un tableau de dimension 2 c'est-à-dire une matrice (8 pages)
- Comment en C passer d'une matrice à un vecteur et réciproquement (7 pages)
- Un exemple simple où l'ordre des opérations dans une addition influe sur le résultat ! (2 pages)
- Représentation selon la Norme (dite) IEEE (de son vrai nom IEC 60559, connue aussi par son ancien nom de IEEE 754 – 1985) du nombre décimal 0.1 (2 pages)
- Quelques exercices de sensibilisation aux problèmes numériques (2 pages)
- Du danger de l'utilisation des opérateurs << et >> (2 pages)
- En annexe : un résumé de la Norme dite IEEE (4 pages)

Page blanche

LES PROBLÈMES DE PORTABILITÉ DU C

Il existe 2 problèmes principaux à la portabilité des applications écrites en C, cela dû en partie à une insuffisance de contraintes dans la Norme (raisons historiques principalement).

1) Le type“ *int* ”:

Rien ne dit quelle est la taille (en unité de *char* , retournée par la fonction *sizeof*) correspondant à une variable de ce type. Dans le tableau suivant, on y indique cette taille pour différents systèmes (le mot « système » représente ici le couple « ordinateur et compilateur »). Dans ce même tableau, on y indique pour des comparaisons, la taille des “ *short int* ” et des “ *long int* ”. On y a fait aussi figurer la taille des données de type “ *size_t* ” , “ *clock_t* ” et “ *time_t* ”. À remarquer que “ *int* ” n’est pas nécessairement synonyme de “ *short int* ” ou de “ *long int* ” .

	short int	*int*	*long int*	*size_t*	*clock_t*	*time_t*	type de *char*
PC CVI	2	4	4	4	4	4	signed
PC Borland	2	2	4	2	4	4	signed
SGI *mode 32bits*	2	4	4	4	4	4	unsigned
SGI *mode 64 bits*	2	4	8	4	4	4	unsigned
Dec-Alpha	2	4	8	8	4	4	signed

2) Le caractère signé ou non-signé du type“ *char* ”:

Rien ne dit si le type “ *char* ” est considéré, par défaut, par le système, comme une valeur signée ou non-signée. On voit dans le tableau quel sont les choix par défaut des mêmes systèmes que précédemment.

Remarque : **Par définition, *sizeof*(*char*) = 1** ; ce qui ne veut pas forcement dire que *char* est toujours représenté par un seul octet !

Page blanche

De la bonne utilisation des prototypes

La "bonne" utilisation de `#include` en ce qui concerne les prototypes

```
/*
D'abord le fichier une_fonc.h  qui permet la déclaration de la fonction une_fonc , fonction
qui ne fait, ici, que transformer son argument, un long int en son équivalent float .
*/

float une_fonc ( long int ) ;


/*
Puis, voici le code du programme principal.
*/

#include <stdio.h>              // pour printf
#include "une_fonc.h"           // pour la déclaration de la fonction une_fonc

void main (void)
{
      long int i = 10 ;
      float x = 3.14f , y ;     // y n'est pas initialisée !

/*
Impression des résultats avant l'appel de la fonction.
On vérifie bien qu'une variable non initialisée contient bien n'importe quoi !
*/
      printf( " avant : i = %d , x = %g , y = %g \n", i, x, y ) ;

/*
Appel à la fonction.
Impression des résultats après l'appel de la fonction.
*/
      y = une_fonc( i ) ;

      printf( " apres : i = %d , x = %g , y = %g \n", i, x, y ) ;
}

/*
Et voici la définition de la fonction. D'abord une version correcte : non seulement, on ne s'est pas
(volontairement) trompé dans la liste des arguments, mais aussi on n'a pas oublié de bien rajouter
l'inclusion du fichier servant à la déclaration de cette même fonction.
*/

#include "une_fonc.h"           //  ICI, noter la présence de cet include

float une_fonc ( long int i )  //  ICI, l'argument est bien de type  long int
{
    return( (float)i ) ;        //  simple transformation de la valeur de l'argument.
}
```

```
/*
Et voici les résultats (la première valeur affichée pour  y  dépend de la machine utilisée) :
*/

 avant : i = 10 , x = 3.14 , y = -1.07374e+008
 apres : i = 10 , x = 3.14 , y = 10

/*
```

C'est bien correct !!

ET maintenant, introduisons une erreur dans la définition de la fonction, en laissant bien, comme il est conseillé de le faire, la ligne `#include` **:**

```
*/

#include "une_fonc.h"

float une_fonc ( float i )
{
    return( (float)i ) ;
}

/*
Pas de problème, le compilateur voit une erreur ...
*/
Warning C4028: formal parameter 1 different from declaration



/*
```

OK MAIS MAINTENANT, SUPPRIMONS LA LIGNE `#include` **:**

```
*/

float une_fonc ( float i )
{
    return( (float)i ) ;
}

/*
Compilons, faisons l'édition des liens (link) : aucun message !
```

On pourrait être satisfait

MAIS

voyons ce qui se passe lors de l'exécution :

```
*/

 avant : i = 10 , x = 3.14 , y = -1.07374e+008
 apres : i = 10 , x = 3.14 , y = 1.4013e-044

/*
L'appel de la fonction (et son retour) fait n'importe quoi !!
*/
```

Comment ‘bien’ déclarer et utiliser en C un tableau de dimension 2 (matrice)

```
#include <stdio.h>                  // pour printf, puts, putchar
#include <stdlib.h>                 // pour malloc, free

#define NDIM1  3                    // par exemple
#define NDIM2  5                    // par exemple
```

/*
Voici une routine d’affichage utilisable dans le cas d’un dimensionnement statique. En C, on peut seulement omettre la première dimension des tableaux c’est -à-dire qu’il faut impérativement que la fonction connaisse les autres dimensions. C’est pour cela que cette fonction n’est utilisable que dans le premier cas.
*/

```
void affiche_1 ( int T[][NDIM2] , int n1 , int n2 )
{
   int i, j ;

   putchar( ’\n’ ) ;
   for ( i = 0 ; i < n1 ; ++i )
   {
      for ( j = 0 ; j < n2 ; ++j )
         printf( " i = %3d ,  j = %3d   ===>  T1 = %5d \n",
                 i, j, T[i][j] ) ;
   }
   putchar( ’\n’ ) ;
}
```

/*
Voici une routine d’affichage utilisable dans le cas d’un dimensionnement dynamique d’u n objet considéré comme une matrice.
*/

```
void affiche_2 ( int * * T , int n1 , int n2 )
{
   int i, j ;

   putchar( ’\n’ ) ;
   for ( i = 0 ; i < n1 ; ++i )
   {
      for ( j = 0 ; j < n2 ; ++j )
      printf( " i = %3d ,  j = %3d   ===>  T2 = %5d \n",
              i, j, T[i][j] ) ;
   }
   putchar( ’\n’ ) ;
}
```

/*
Voici une routine d'affichage utilisable dans le cas d'un dimensionnement dynamique d'un objet considéré comme un vecteur.
*/

```
void affiche_3 ( int * V , int n1 , int n2 )
{
   int i, j, k ;

   putchar( '\n' ) ;
   for ( i = 0 ; i < n1 ; ++i )
   {
      for ( j = 0 ; j < n2 ; ++j )
      {
          k = i * n2 + j ;              // calcul de l'adresse
          printf( " i = %3d ,  j = %3d ,  k = %3d   ===>  V = %5d \n",
                  i, j, k, V[k] ) ;
      }
   }
   putchar( '\n' ) ;
}
```

/*
Voici, après les routines d'affichage (placées en tête que pour des raisons de compilation), le programme principal avec les différentes possibilités et les explication s qui vont avec.

Le cas '1' présente une allocation entièrement statique de la matrice. Dans le programme principal donné en illustration, le tableau déclaré ainsi aura comme nom `ANZ` .

Le cas '2' présente une allocation entièrement dynamique de la matri ce i.e. chaque ligne '`T[i]`' est (réellement) indépendante des autres et donc il n'y a aucune raison que le stockage en mémoire des éléments de la matrice soit contigu (en revanche, chaque ligne peut ainsi être allouée avec une taille différente mais c'est une autre application que celle, ici, de déclarer une matrice). En ce qui concerne les matrices triangulaires ou en bande, même si cette solution est admissible, la solution présentée au cas suivant, moyennant bien sûr une adaptation, est bien meilleure. D ans le programme principal donné en illustration, le tableau déclaré ainsi aura comme nom `BP` .

Le cas '3' présente une allocation dynamique mais où on préserve la contiguïté des éléments. C'est donc la 'solution' préconisée et pour apprendre à s'en servir, on se reportera aux explications données dans le programme principal. Dans ce programme principal, tableau déclaré ainsi aura comme nom `PIC` .
*/

```
void main (void)
{
   int i, j, n1 = NDIM1 , n2 = NDIM2 ;

   int ANZ[NDIM1][NDIM2] ;              // dimensionnement statique
   int * * BP ;                         // dimensionnement dynamique
   int * * PIC ;                        // dimensionnement dynamique
```

/* **CAS STATIQUE** (lignes contiguës) */

```
   puts( " Cas 1 : ANZ mode \n" ) ;
```

/*
On regarde les tailles dans le cas du dimensionnement statique
*/

```
   printf( " sizeof de ANZ : %d \n", sizeof(ANZ) ) ;
   printf( " taille de ANZ : %d \n", sizeof(ANZ)/sizeof(**ANZ) ) ;
   putchar( '\n' ) ;
```

/*
On peut donc remplir de suite le tableau (par exemple, avec un incrément de 10 par ligne)
*/

```
   for ( i = 0 ; i < NDIM1 ; ++i )
   {
      for ( j = 0 ; j < NDIM2 ; ++j ) ANZ[i][j] = 10 * i + j ;
   }
```

/*
Et regarder ce qu'il y a dedans
*/

```
   affiche_1 ( ANZ , NDIM1 , NDIM2 ) ;
// affiche_2 ( (int * *)ANZ , n1 , n2 ) ;        // run time error ? !
   affiche_3 ( (int *)ANZ , n1 , n2 ) ;

   puts( " ================================================ \n" ) ;
```

/* **CAS DYNAMIQUE** (lignes non nécessairement contiguës) */

```
   puts( " Cas 2 : BP  mode \n" ) ;
```

/*
On alloue d'abord le tableau contenant les pointeurs des lignes (bien voir la taille demandée)
*/

```
   BP = (int * *) malloc( n1 * sizeof( int * ) ) ;
   if ( NULL == BP ) puts( " PB1 " ) ;
```

/*
Puis, on alloue chaque ligne à la longueur souhaitée mais on remarque que chaque ligne est bien totalement indépendante des autres.
*/

```
   for ( i = 0 ; i < n1 ; ++i )
   {
      BP[i] = ( int * ) malloc( n2 * sizeof(int) ) ;
      if ( NULL == BP[i] ) printf( " PB2 : i = %d \n", i ) ;
   }
```

/*
On peut alors regarder les tailles, en fait maintenant celles des pointeurs.
*/

```
   printf( " sizeof de  BP : %d \n", sizeof(BP) ) ;
   printf( " sizeof de *BP : %d \n", sizeof(*BP) ) ;
   putchar( '\n' ) ;
```

/*
On peut donc alors remplir le tableau (par exemple, avec un incrément de 100 par ligne)
*/

```
   for ( i = 0 ; i < n1 ; ++i )
   {
      for ( j = 0 ; j < n2 ; ++j ) BP[i][j] = 100 * i + j ;
   }
```

/*
Et regarder de qu'il y a dedans. Selon les ordinateurs, on peut constater, que l'affichage par la routine 'affiche_3' qui suppose que les éléments sont réellement contigus, peut donner n'importe quoi.
*/

```
   affiche_2 ( BP , n1 , n2 ) ;
   affiche_3 ( *BP , n1 , n2 ) ;
```

/*
Et ne pas oublier, si l'on veut être propre de libérer la mémoire le plus tôt possible i.e. ne pas attendre la fin du programme. On le fait, bien sûr, dans l'ordre inverse de celui de l'allocation.
*/

```
   for ( i = 0 ; i < n1 ; ++i ) free( BP[i] ) ;
   free( BP ) ;

   puts( " ================================================ \n" ) ;
```

/* **CAS DYNAMIQUE** (lignes contiguës) */

```
   puts( " Cas 3 : PIC  mode \n" ) ;
```

/*
On alloue d'abord le tableau contenant les pointeurs des lignes (bien voir la taille demandée).
Là, pas de changement par rapport au cas précédent.
*/

```
   PIC = (int * *) malloc( n1 * sizeof( int * ) ) ;
   if ( NULL == PIC ) puts( " PB3 " ) ;
```

```
/*
Puis, on alloue pour la première ligne la taille totale de la matrice (bien voir la taille demandée).
Après, maintenant que la totalité de l'espace de stockage consacrée à la matrice a bien été allouée,
on le partitionne selon les tailles souhaitées à chaque ligne (c'est à cet endroit que l'on peut
introduire les modifications correspondantes aux cas des matrices triangulaires ou des matrices
bande). On se sert pour cela des propriétés de l'arithmétique des pointeurs.
*/

   PIC[0] = (int *) malloc( n1 * n2 * sizeof(int) ) ;
   if ( NULL == PIC[0] ) puts( " PB4 " ) ;
   for ( i = 1 ; i < n1 ; ++i ) PIC[i] = PIC[i-1] + n2 ;


/*
On peut alors regarder les tailles, en fait maintenant celles des pointeurs.
*/

   printf( " sizeof de  PIC : %d \n", sizeof(PIC)  ) ;
   printf( " sizeof de *PIC : %d \n", sizeof(*PIC) ) ;
   putchar( '\n' ) ;


/*
On peut donc alors remplir le tableau (par exemple, avec un incrément d e 1000 par ligne)
*/

   for ( i = 0 ; i < n1 ; ++i )
   {
      for ( j = 0 ; j < n2 ; ++j ) PIC[i][j] = 1000 * i + j ;
   }


/*
Et regarder ce qu'il y a dedans.
*/

   affiche_2 ( PIC , n1 , n2 ) ;
   affiche_3 ( PIC[0] , n1 , n2 ) ;
   puts( " ================================================ \n" ) ;


/*
Et ne pas oublier, si l'on veut être propre de libérer la mémoire le plus tôt possible i.e. ne pas
attendre la fin du programme. On le fait, bien sûr, dans l'ordre inverse de celui de l'allocati on.
*/

   free( *PIC ) ;
   free( PIC ) ;


   puts( " \n C'est fini ! \n" ) ;
}
```

Et voici les sorties de ce programme :

```
Cas 1 : ANZ mode

 sizeof de ANZ : 60
 taille de ANZ : 15

 i =   0 ,  j =   0   ===>  T1 =     0
 i =   0 ,  j =   1   ===>  T1 =     1
 i =   0 ,  j =   2   ===>  T1 =     2
 i =   0 ,  j =   3   ===>  T1 =     3
 i =   0 ,  j =   4   ===>  T1 =     4
 i =   1 ,  j =   0   ===>  T1 =    10
 i =   1 ,  j =   1   ===>  T1 =    11
 i =   1 ,  j =   2   ===>  T1 =    12
 i =   1 ,  j =   3   ===>  T1 =    13
 i =   1 ,  j =   4   ===>  T1 =    14
 i =   2 ,  j =   0   ===>  T1 =    20
 i =   2 ,  j =   1   ===>  T1 =    21
 i =   2 ,  j =   2   ===>  T1 =    22
 i =   2 ,  j =   3   ===>  T1 =    23
 i =   2 ,  j =   4   ===>  T1 =    24

 i =   0 ,  j =   0 ,  k =   0   ===>  V =     0
 i =   0 ,  j =   1 ,  k =   1   ===>  V =     1
 i =   0 ,  j =   2 ,  k =   2   ===>  V =     2
 i =   0 ,  j =   3 ,  k =   3   ===>  V =     3
 i =   0 ,  j =   4 ,  k =   4   ===>  V =     4
 i =   1 ,  j =   0 ,  k =   5   ===>  V =    10
 i =   1 ,  j =   1 ,  k =   6   ===>  V =    11
 i =   1 ,  j =   2 ,  k =   7   ===>  V =    12
 i =   1 ,  j =   3 ,  k =   8   ===>  V =    13
 i =   1 ,  j =   4 ,  k =   9   ===>  V =    14
 i =   2 ,  j =   0 ,  k =  10   ===>  V =    20
 i =   2 ,  j =   1 ,  k =  11   ===>  V =    21
 i =   2 ,  j =   2 ,  k =  12   ===>  V =    22
 i =   2 ,  j =   3 ,  k =  13   ===>  V =    23
 i =   2 ,  j =   4 ,  k =  14   ===>  V =    24

 ================================================

 Cas 2 : BP  mode

 sizeof de  BP : 4
 sizeof de *BP : 4

 i =   0 ,  j =   0   ===>  T2 =     0
 i =   0 ,  j =   1   ===>  T2 =     1
 i =   0 ,  j =   2   ===>  T2 =     2
 i =   0 ,  j =   3   ===>  T2 =     3
 i =   0 ,  j =   4   ===>  T2 =     4
 i =   1 ,  j =   0   ===>  T2 =   100
 i =   1 ,  j =   1   ===>  T2 =   101
 i =   1 ,  j =   2   ===>  T2 =   102
 i =   1 ,  j =   3   ===>  T2 =   103
 i =   1 ,  j =   4   ===>  T2 =   104
 i =   2 ,  j =   0   ===>  T2 =   200
 i =   2 ,  j =   1   ===>  T2 =   201
```

```
i =   2 ,  j =   2   ===>  T2 =   202
i =   2 ,  j =   3   ===>  T2 =   203
i =   2 ,  j =   4   ===>  T2 =   204

i =   0 ,  j =   0 ,  k =   0   ===>  V =     0
i =   0 ,  j =   1 ,  k =   1   ===>  V =     1
i =   0 ,  j =   2 ,  k =   2   ===>  V =     2
i =   0 ,  j =   3 ,  k =   3   ===>  V =     3
i =   0 ,  j =   4 ,  k =   4   ===>  V =     4
i =   1 ,  j =   0 ,  k =   5   ===>  V = -33686019
i =   1 ,  j =   1 ,  k =   6   ===>  V = 524296
i =   1 ,  j =   2 ,  k =   7   ===>  V = 524544
i =   1 ,  j =   3 ,  k =   8   ===>  V = 3082168
i =   1 ,  j =   4 ,  k =   9   ===>  V = 3082296
i =   2 ,  j =   0 ,  k =  10   ===>  V =     0
i =   2 ,  j =   1 ,  k =  11   ===>  V =     0
i =   2 ,  j =   2 ,  k =  12   ===>  V =    20
i =   2 ,  j =   3 ,  k =  13   ===>  V =     1
i =   2 ,  j =   4 ,  k =  14   ===>  V =    46

================================================

Cas 3 : PIC  mode

sizeof de  PIC : 4
sizeof de *PIC : 4

i =   0 ,  j =   0   ===>  T2 =     0
i =   0 ,  j =   1   ===>  T2 =     1
i =   0 ,  j =   2   ===>  T2 =     2
i =   0 ,  j =   3   ===>  T2 =     3
i =   0 ,  j =   4   ===>  T2 =     4
i =   1 ,  j =   0   ===>  T2 =  1000
i =   1 ,  j =   1   ===>  T2 =  1001
i =   1 ,  j =   2   ===>  T2 =  1002
i =   1 ,  j =   3   ===>  T2 =  1003
i =   1 ,  j =   4   ===>  T2 =  1004
i =   2 ,  j =   0   ===>  T2 =  2000
i =   2 ,  j =   1   ===>  T2 =  2001
i =   2 ,  j =   2   ===>  T2 =  2002
i =   2 ,  j =   3   ===>  T2 =  2003
i =   2 ,  j =   4   ===>  T2 =  2004

i =   0 ,  j =   0 ,  k =   0   ===>  V =     0
i =   0 ,  j =   1 ,  k =   1   ===>  V =     1
i =   0 ,  j =   2 ,  k =   2   ===>  V =     2
i =   0 ,  j =   3 ,  k =   3   ===>  V =     3
i =   0 ,  j =   4 ,  k =   4   ===>  V =     4
i =   1 ,  j =   0 ,  k =   5   ===>  V =  1000
i =   1 ,  j =   1 ,  k =   6   ===>  V =  1001
i =   1 ,  j =   2 ,  k =   7   ===>  V =  1002
i =   1 ,  j =   3 ,  k =   8   ===>  V =  1003
i =   1 ,  j =   4 ,  k =   9   ===>  V =  1004
i =   2 ,  j =   0 ,  k =  10   ===>  V =  2000
i =   2 ,  j =   1 ,  k =  11   ===>  V =  2001
i =   2 ,  j =   2 ,  k =  12   ===>  V =  2002
i =   2 ,  j =   3 ,  k =  13   ===>  V =  2003
i =   2 ,  j =   4 ,  k =  14   ===>  V =  2004

================================================
```

C'est fini !

Comment passer d'une matrice à un vecteur et réciproquement

```
/*
En ce qui concerne la 'bonne' déclaration d'une matrice (telle qu'utilisée ici), le lecteur se reportera
à l'autre document nommé « Comment 'bien' déclarer et utiliser en C un tableau de dimension 2 ».
Dans ce document, on ne rappellera que rapidement comment il faut faire et le lecteur verra que,
selon les cas, la meilleure façon restera de travailler avec seulement un vecteur (voir à la fin).
*/


#include <stdio.h>            // pour printf, puts
#include <stdlib.h>           // pour malloc, free


/*
ICI, le programme principal  main  renvoie des codes de bon/mauvais fonctionnement ; il s'agit
des valeurs EXIT_SUCCESS et EXIT_FAILURE définies dans stdlib.h .
*/

int main (void)
{


/*
On introduit une première instruction  typedef  afin de pouvoir changer plus facilement de type
de données traitées par ce programme. Par exemple, si l'on veut 'convertir' ce programme pour
qu'il fonctionne avec des entiers ( short int  par exemple), il suffit de modifier qu'une seule
ligne en remplaçant typedef double TYPE par typedef short int TYPE .
Le lecteur remarquera que, pour que le programme puisse être correct sans autre modification, il a
fallu compliquer un tout petit peu les instructions de visualisation des données.
De même, les deux autres instructions  typedef  servent à introduire (c'est de la meilleure
pratique que d'introduire ainsi des nouveaux tpyes) les deux autres types manipulés par ce
programme, à savoir, le type  VECTOR  et le type  MATRIX  construits à partir de  TYPE
précédemment défini.
*/

   typedef double TYPE ;
// typedef float TYPE ;
// typedef long int TYPE ;
// typedef short int TYPE ;
// typedef unsigned char TYPE ;
// ...
```

```
typedef TYPE * VECTOR ;
typedef TYPE * * MATRIX ;
```

/*
Ici, on se définit `dim` comme étant la longueur du vecteur ; cette longueur devant correspondre exactement (c'est mieux !) au nombre de cases de la matrice correspondante, on définit donc la valeur de `dim` à partir de celles de `dim1` et `dim2` , valeurs du nombre de lignes et de colonnes de la matrice.
De plus, à titre pédagogique, pour un vecteur de 8 cases (notre exemple), on se définit quatre possibilités de voir un tel vecteur soit comme une matrice 2 x 4 , soit comme une matrice 4 x 2 ; Noter aussi les possibilités d'avoir des matrices 1 x 8 ou 8 x 1 (ce qui n'est pas équivalent à un vecteur …).
*/

```
// int const dim1 = 8 , dim2 = 1 ;
   int const dim1 = 4 , dim2 = 2 ;
// int const dim1 = 2 , dim2 = 4 ;
// int const dim1 = 1 , dim2 = 8 ;

   int const dim = dim1 * dim2 ;


   int i, j ;                    // les indices des boucles
```

/*
On déclare :

- d'une part, la matrice de départ `m_in` et le vecteur qui lui sera équivalent `v_out` utilisés dans la première partie de notre exemple (on commence toujours par le plus facile …)
- d'autre part, le vecteur de départ `v_in` et la matrice que l'on voudra lui rendre équivalente `m_out` utilisés dans la seconde partie de notre exemple (il y a plus de travail à faire …).

*/

```
   VECTOR v_out , v_in ;
   MATRIX m_in , m_out ;


   puts ( " Premiere partie : Matrice ==> Vecteur \n" ) ;
```

/*
Dans cette première partie, on part d'une matrice construite avec des cases contiguës `m_in` .
Le but poursuivi est de créer un alias de la matrice (c'est-à-dire ayant la même adresse et, bien sûr, sans réservation de mémoire supplémentaire) qui soit vu comme un (simple) vecteur `v_out` .
*/

/*
Dans un premier temps, on va donc créer cette matrice et la remplir par des valeurs reconnaissables. Ceci est fait en allouant d'abord, pour l'objet de type `MATRIX` , un vecteur destiné à recevoir les 'pointeurs' des lignes, puis en allouant la mémoire nécessaire aux éléments de la matrice pour le premier pointeur de ligne i.e. `m_in[0]` soit encore `*m_in` ; il ne reste plus qu'à indiquer que les autres pointeurs de lignes correspondent au début des autres lignes de la matrice : c'est le rôle de la boucle `for` . Remarquer que l'on se sert dans la boucle `for` d'une des propriétés fondamentales de l'arithmétique des pointeurs. (Pour plus de détails, voir le document déjà signalé au début).
*/

```
   m_in = (MATRIX) malloc( dim1 * sizeof(VECTOR) ) ;
   if ( NULL == m_in ) return(EXIT_FAILURE) ;

   *m_in = (VECTOR) malloc( dim * sizeof(TYPE) ) ;
   if ( NULL == *m_in ) return(EXIT_FAILURE) ;

   for ( i = 1 ; i < dim1 ; ++i ) m_in[i] = m_in[i-1] + dim2 ;


   for ( i = 0 ; i < dim1 ; ++i)
   {
      for ( j = 0 ; j < dim2 ; ++j ) m_in[i][j] = (TYPE)(10*i+j) ;
   }

   for ( i = 0 ; i < dim1 ; ++i)
   {
      for ( j = 0 ; j < dim2 ; ++j )
      printf( " m_in[%d][%d] = %g \n" , i, j, (double)m_in[i][j] ) ;
   }
   puts(" ") ;
```

/*
On va maintenant dire que le vecteur `v_out` est équivalent à la matrice `m_in` . Comme les cases mémoires sont déjà allouées, il suffit de donner comme valeur à ce pointeur la valeur qui lui convient ! Nous voyons, à la fois pour des raisons de correspondance de type (chaque entité de part et d'autre de l'affectation doivent être du même type, ici `TYPE *`) et pour des raisons de taille d'allocation (c'est bien à `*m_in` que l'on a alloué la taille nécessaire au stockage des données), que la seule possibilité est celle qui est indiquée. Remarquons toutefois que si la matrice avait été allouée de manière statique, l'indirection n'étant pas possible, on aurait été obligé d'écrire l'affectation autrement : `v_out = &m_in[0][0] ;`
*/

```
   v_out = *m_in ;                          // tout est là


   puts( " On fait les verifications : " ) ;

   printf( " pour m_in  : %p \n", m_in  ) ;
   printf( " pour *m_in : %p \n", *m_in ) ;
   printf( " pour v_out : %p \n", v_out ) ;

   puts ( " OK pour les adresses \n" ) ;
```

```
   for ( i = 0 ; i < dim ; ++i )
      printf( " v_out[%d] = %g \n" , i, (double)v_out[i] ) ;
   puts ( " OK aussi pour les valeurs \n" ) ;


/*
Même si ce n'est pas strictement nécessaire dans le programme principal, on n'oublie pas de
désallouer  proprement ce qui a été alloué dynamiquement.
*/

   free( *m_in ) ;
   free( m_in ) ;

   puts( " FIN DE LA PREMIERE PARTIE \n\n" ) ;


/*
```

Et maintenant, on va essayer de faire l'inverse !!

```
À savoir, en partant d'un vecteur  v_in  (dont les cases seront donc, par construction, contiguës),
on veut créer un alias (c'est-à-dire ayant la même adresse mais, comme on le verra par la suite et
contrairement au premier cas, au prix d'une réservation de mémoire supplémentaire) qui soit vu
comme une matrice  m_out  ,  au choix 8x1, 4x2, 2x4 ...
*/

   puts ( " Seconde partie : Vecteur ==> Matrice \n" ) ;


/*
On alloue d'abord les cases du vecteur puis on les remplit avec des valeurs reconnaissables.
*/

   v_in = malloc( dim * sizeof(TYPE) ) ;
   if ( NULL == v_in ) return(EXIT_FAILURE) ;

   for ( i = 0 ; i < dim ; ++i ) v_in[i] = (TYPE)(i+100) ;

   for ( i = 0 ; i < dim ; ++i )
        printf( " v_in[%d] = %g \n" , i, (double)v_in[i] ) ;
   puts (" ") ;


/*
Comme on veut travailler au final avec une matrice, on n'a pas d'autre choix que de créer (ce qui
implique l'utilisation de mémoire supplémentaire) comme dans le premier cas, un vecteur destiné à
recevoir les 'pointeurs' des lignes de la (future) matrice.
*/

   m_out = (MATRIX) malloc( dim1 * sizeof(VECTOR) ) ;
   if ( NULL == m_out ) return(EXIT_FAILURE) ;
```

```
/*
Il ne reste plus qu'à donner la bonne valeur (l'adresse du vecteur v_in ) au premier 'pointeur' de
lignes de la matrice ; remarquons, comme précédemment que c'est bien *m_out qui est bien du
même type que v_in , à savoir TYPE * ).
*/

   *m_out = v_in ;


/*
Sans faire l'allocation de m_out , on aurait pu penser qu'il était suffisant de faire :
   m_out = (MATRIX)v_in ;    *m_out = v_in ;
ce qui implique, en ce qui concerne la valeur des adresses, que non seulement celle de *m_out
est la même que celle de v_in (normal) mais aussi celle de m_out . Malheureusement, du fait
de l'absence d'allocation pour les 'pointeurs' des lignes, ceux-ci seront donc stockés dans les
premières cases de m_out , cet objet ayant la même adresse que *m_out , écrasant par là les
valeurs des premières cases du vecteur v_in (correspondantes à la première ligne de la matrice).
Faire l'expérience pour voir !!

Et comme dans le cas précédent, il faut indiquer que les autres pointeurs de lignes correspondent au
début des différentes lignes de la matrice. Remarquer que l'on se sert dans la boucle for d'une
des propriétés fondamentales de l'arithmétique des pointeurs.
*/

   for ( i = 1 ; i < dim1 ; ++i ) m_out[i] = m_out[i-1] + dim2 ;


   puts( " On fait les verifications : " ) ; //   quelques vérifications !

   printf( " pour v_in     : %p \n", v_in     ) ;
   printf( " pour m_out    : %p \n", m_out    ) ;
   printf( " pour *m_out   : %p \n", *m_out   ) ;
   printf( " pour m_out[0] : %p \n", m_out[0] ) ;      // idem !

   puts (" OK pour les adresses \n") ;


   for ( i = 1 ; i < dim1 ; ++i )
      printf( " pour m_out[%d] : %p \n", i, m_out[i] ) ;

   if ( dim1 > 0 )
      puts (" OK aussi pour les adresses attendues \n") ;


   for ( i = 0 ; i < dim1 ; ++i)
   {
      for ( j = 0 ; j < dim2 ; ++j )
      printf( " m_out[%d][%d] = %g \n" , i, j, (double)m_out[i][j] ) ;
   }

   puts (" OK aussi pour les valeurs \n") ;
```

```
/*
Même si ce n'est pas strictement nécessaire dans le programme principal, on n'oublie pas de
désallouer  proprement ce qui a été alloué dynamiquement.
*/

   free( v_in ) ;
   free( m_out ) ;

   puts( " FIN DE LA SECONDE PARTIE \n\n" ) ;


   return(EXIT_SUCCESS) ;
}
```

Remarques finales :

Au total, nous voyons que l'utilisation d'une matrice (tableau à 2 dimensions) consomme toujours plus de mémoire qu'un seul vecteur de taille (mathématique) identique.
Aussi et même si les cases de la matrice sont contiguës (je n'imagine pas d'ailleurs que l'on puisse, pour des matrices carrées, rectangulaires et même triangulaires ou à bande, faire autrement !), l'usage d'un tableau à 2 dimensions requiert toujours 2 indirections (c'est l'opérateur `*` ou sa version cachée en `[..][..]`) ce qui consomme aussi un peu plus de temps de calcul dans les accès mémoire.
Ceci dit, un des gros avantages de l'utilisation des matrices en tant que telles, reste la proximité entre l'objet mathématique (et les utilisations que l'on en fait) et sa version informatique. Si cela était demandé (en exercice, par exemple) de faire une procédure de résolution d'un système linéaire par la méthode du (plus grand) pivot en n'utilisant que des objets (informatiques) de type vecteur, le code écrit serait bien moins facile à lire et à construire que sa version naturelle en utilisant des matrices.

Remerciements à Alain NoulleZ pour sa participation à la résolution de ce problème.

Et voici, les résultats du programme (bien sûr les sorties obtenues pour la valeur des pointeurs dépendent du système utilisé).

```
Premiere partie : Matrice ==> Vecteur

 m_in[0][0] = 0
 m_in[0][1] = 1
 m_in[1][0] = 10
 m_in[1][1] = 11
 m_in[2][0] = 20
 m_in[2][1] = 21
 m_in[3][0] = 30
 m_in[3][1] = 31

 On fait les verifications :
 pour m_in  : 002F07A8
 pour *m_in : 002F07F0
 pour v_out : 002F07F0
 OK pour les adresses
```

```
v_out[0] = 0
v_out[1] = 1
v_out[2] = 10
v_out[3] = 11
v_out[4] = 20
v_out[5] = 21
v_out[6] = 30
v_out[7] = 31
OK aussi pour les valeurs

FIN DE LA PREMIERE PARTIE


Seconde partie : Vecteur ==> Matrice

v_in[0] = 100
v_in[1] = 101
v_in[2] = 102
v_in[3] = 103
v_in[4] = 104
v_in[5] = 105
v_in[6] = 106
v_in[7] = 107

On fait les verifications :
pour v_in     : 002F07A8
pour m_out    : 002F0820
pour *m_out   : 002F07A8
pour m_out[0] : 002F07A8
OK pour les adresses

pour m_out[1] : 002F07B8
pour m_out[2] : 002F07C8
pour m_out[3] : 002F07D8
OK aussi pour les adresses attendues

m_out[0][0] = 100
m_out[0][1] = 101
m_out[1][0] = 102
m_out[1][1] = 103
m_out[2][0] = 104
m_out[2][1] = 105
m_out[3][0] = 106
m_out[3][1] = 107
OK aussi pour les valeurs

FIN DE LA SECONDE PARTIE
```

Page blanche

Quand l'addition n'est pas associative, ni commutative

Rappel de l'énoncé

Calculer exactement comme cela est indiqué (en SP comme en DP), les quantités suivantes)
pour : a = 0.05 , b = 0.2825 , c = 0.65 , d = 0.0175
t1 = a + b + c , t2 = a + b + d , x1 = t1 + d , x2 = t2 + c
Comparaison de x1 et x2, Conclusion, explications ?

Voici le programme écrit en C

```
#include <stdio.h>

void main (void)
{
   typedef float  SP ;
   typedef double DP ;

   SP sa = 0.05f , sb = 0.2825f , sc = 0.65f , sd = 0.0175f , st1, st2, sx1, sx2 ;
   DP da = 0.05  , db = 0.2825  , dc = 0.65  , dd = 0.0175  , dt1, dt2, dx1, dx2 ;

   puts(" en principe, la somme doit faire UN ! \n") ;

   st1 = sa + sb + sc ;
   st2 = sa + sb + sd ;
   sx1 = st1 + sd ;
   sx2 = st2 + sc ;

   dt1 = da + db + dc ;
   dt2 = da + db + dd ;
   dx1 = dt1 + dd ;
   dx2 = dt2 + dc ;

   printf(" %.7f    %.18lf \n"   , sx1, dx1) ;
   printf(" %.7f    %.18lf \n\n" , sx2, dx2) ;

   puts(" et le reste par rapport a 1. ZERO ! \n") ;
   printf(" %.7f    %.18lf \n"   , sx1 - 1.f , dx1 - 1. ) ;
   printf(" %.7f    %.18lf \n\n" , sx2 - 1.f , dx2 - 1. ) ;

   puts( " La ou devrait avoir 4 fois la reponse ' Vrai : gagne ! ' " ) ;
   puts( " ====> voir le programme et les resultats !!!! \n" ) ;

   if ( sx1 == 1. ) puts(" Vrai : gagne ! ") ;
   else             puts(" Faux : perdu ! ") ;

   if ( sx2 == 1. ) puts(" Vrai : gagne ! ") ;
   else             puts(" Faux : perdu ! ") ;

   if ( dx1 == 1. ) puts(" Vrai : gagne ! ") ;
   else             puts(" Faux : perdu ! ") ;

   if ( dx2 == 1. ) puts(" Vrai : gagne ! ") ;
   else             puts(" Faux : perdu ! ") ;

   puts( " " ) ;
}
```

Et voici ***une*** réponse

```
en principe, la somme doit faire UN !

 0.9999999    0.999999999999999890
 1.0000000    1.000000000000000000

 et le reste par rapport a 1. ZERO !

 -0.0000001    -0.000000000000000111
  0.0000000     0.000000000000000000

 La ou devrait avoir 4 fois la reponse ' Vrai : gagne ! '
 ====> voir le programme et les resultats !!!!

 Faux : perdu !
 Vrai : gagne !
 Faux : perdu !
 Vrai : gagne !
```

Au sujet d'une conversation sur les nombres réels écrits en Simple Précision (SP) ou en Double Précision (DP) dans un programme, voici une petite démonstration des résultats que cela peut donner lorsque l'on ne fait pas attention à ce que l'on croit n'être que maniaqueries de numériciens ! (cette note de cours améliore donc grandement mon exemple de cours au DEA d'Astrophysique).

On se place dans la représentation IEEE des nombres flottants du type '*big endian*' c'est-à-dire d'abord le bit de signe (souligné par le caractère '|'), puis l'exposant (biaisé) (dont les bits sont soulignés par les caractères '^^^^^^') puis enfin la mantisse (dans son ordre naturel).

L'exemple est construit autour des valeurs *machine* du nombre décimal `0.1` qui, il faut le remarquer, n'est pas un nombre représentable exactement en binaire comme on va le voir dans la partie suivante.

D'abord des résultats mathématiquement exacts :

$$0.1 = 9\sum_{i=1}^{\infty}\left(\tfrac{1}{16}\right)^{i} - 0.5$$

d'où la représentation binaire (infinie) de `0.1` :

$$0.1_{10} = 0.0001\ 1001\ 1001\ 1001\ 1001\ \ldots\ _{2}$$
$$= 0.19999\ \ldots\ _{16}$$

Avec les arrondis IEEE de la SP, `0.1` (Fortran) ou `0.1f` (C) s'écrit donc :

En binaire :
```
00111101110011001100110011001101
|^^^^^^^^
```

ce qui représente la valeur exacte de :

$$0.1(SP) = 0.1 + \tfrac{2}{5}\left(\tfrac{1}{16}\right)^{7}$$

Avec les arrondis IEEE de la DP, `0.1d0` (Fortran) ou `0.1` (C) s'écrit donc :

En binaire :
```
0011111110111001100110011001100110011001100110011001100110011010
|^^^^^^^^^^^
```

ce qui représente la valeur exacte de :

$$0.1(DP) = 0.1 + \tfrac{2}{5}\left(\tfrac{1}{16}\right)^{14}$$

Si on aligne les mantisses des 2 nombres :

```
SP : 10011001100110011001101
DP : 1001100110011001100110011001100110011001100110011010
```

on voit bien que le `0.1(SP)` est un tout petit plus grand que `0.1 (DP)`

On peut aussi comparer l'erreur de représentation et le *spacing* :

L'erreur de représentation est définie par `0.1 - 0.1(P)` où `P` est la précision.

	Erreur de représentation	*Spacing* (au voisinage de 0.1)
SP	≈ 1.490 E-09	≈ 7.451 E-09
DP	≈ 5.551 E-18	≈ 13.88 E-18

A titre d'exemple, voici un programme Fortran mettant en oeuvre ce qui vient d'être dit :

```
Program ZERO_ONE
   Double Precision :: x
   !
   Write(*,*) " 0.1 en SP et en binaire : "
   Write(*,"(B32)" ) 0.1
   Write(*,"(B32)" ) REAL( 0.1d0 - 0.4d0 * 0.0625d0**7 )      ! verification
   Write(*,*)
   !
   Write(*,*) " 0.1 en DP et en binaire : "
   Write(*,"(B64)" ) 0.1d0
   Write(*,"(B64)" ) 0.1d0 - 0.4d0 * 0.0625d0**14        ! verification ===>
   Write(*,*)                                            ! cancellation
   !                                                     !    des bits !!
   Write(*,*) " On se sert de la fonction Spacing pour des comparaisons : "
   Write(*,*) SPACING(0.1), SPACING(0.1d0)
   Write(*,*) 0.4 * 0.0625**7 , 0.4d0 * 0.0625d0**14
   Write(*,*)
   !
   Write(*,*) " Pour aller jusqu'au dernier bit : on peut le faire comme ce"
   Write(*,*)"     qui va suivre car on travaille bien en base 2 "
   x = 2.5d0 * ( 0.1 - 0.1d0 ) * 16.d0**7            ! valeur theorique de x
   Write(*,*) x                                      !   x = 1 - (1/16)^7
   Write(*,*) 1.d0 - 0.0625d0**7                     !   ====> OK
   !
   Write(*,"(A)",Advance="No") "                      et encore mieux : "
   Write(*,*) REAL( ( 1.d0 - x ) * 16.d0**7 )
   Write(*,*)
   !
   STOP " PARFAIT !! "
End Program ZERO_ONE
```

et son listing de sortie :

```
0.1 en SP et en binaire :
  11110111001100110011001101
  11110111001100110011001101

0.1 en DP et en binaire :
 1111111011100110011001100110011001100110011001100110011010
 1111111011100110011001100110011001100110011001100110011010

On se sert de la fonction Spacing pour des comparaisons :
  7.4505806E-09  1.387778780781446E-017
  1.4901161E-09  5.551115123125783E-018

Pour aller jusqu'au dernier bit : on peut le faire comme ce
    qui va suivre car on travaille bien en base 2
0.999999996274710
0.999999996274710
                    et encore mieux :    1.000000

 PARFAIT !!
```

EXERCICES DE SENSIBILISATION AUX PROBLÉMES NUMÉRIQUES

Bernard PICHON

• 1 • Suite de Fibonacci

Soit $F_{n+1} = F_n + F_{n-1}$ avec $F_0 = 0$ et $F_1 = 1$.
Calculer et afficher $W_n - 1$ où $W_n = V_n^2 - V_n$ et $V_n = \frac{F_{n+2}}{F_{n+1}}$

–a– Après le premier essai, essayer de trouver une relation de récurrence concernant directement les W_n . En déduire un test numérique, le faire (trouver aussi la démonstration de cette relation).

–b– Même par la méthode "naive", on peut trouver une relation de récurrence pour les V_n, donc pouvoir les calculer sans avoir à calculer les F_n . Le faire.

–c– Soit $f(x, y) = 2xy^4 + x^2y^3 - 2x^3y^2 - y^5 - x^4y + 2y$
Calculer pour x et y entiers strictement positifs, les valeurs positives prises par cette fonction (on pourra, en réécrivant ce polynôme, faire des encadrements de ses valeurs prises pour différentes valeurs relatives de x et y). Quels souvenirs ? ? (la démonstration *n'est pas* demandée ...).

–d– Rmq : $\forall n, k$, $F_n \mid F_{n\,k}$ Si $(m, n) = d$ alors $(F_m, F_n) = F_d$.

• 2 • Calcul des éléments d'une suite

Soit la suite

$$U_{n+1} = A - \frac{B}{U_n} + \frac{C}{U_n\, U_{n-1}}$$

Calculer numériquement à l'aide d'un programme la limite de cette suite pour les valeurs suivantes des paramètres :

$A = 6$, $B = 11$ et $C = 6$	; $U_0 = 3/2$ et $U_1 = 5/3$
$A = 11$, $B = 36$ et $C = 36$	; $U_0 = 5/2$ et $U_1 = 13/5$
$A = 6$, $B = 3$ et $C = -10$	; $U_0 = 1/2$ et $U_1 = 5$
$A = 111$, $B = 1130$ et $C = 3000$	; $U_0 = 11/2$ et $U_1 = 61/11$
$A = 9$, $B = 23$ et $C = 15$	; $U_0 = 2$ et $U_1 = 5/2$
$A = 8$, $B = 17$ et $C = 10$	; $U_0 = 3/2$ et $U_1 = 5/3$

–a– La "bonne" limite est : $b = U_0 + \sqrt{U_0\, U_1 - U_0^2}$

–b– La "mauvaise" limite est : $c = A - 2\, U_0$

–c– Rmq : Rechercher la solution générale sous la forme $U_n = \frac{a_{n+1}}{a_n}$

• 3 • Non-distributivité ET/OU Non-commutativité

–a– Calculer (en SP comme en DP), les quantités suivantes (et exactement comme cela est indiqué) pour :
$a = 0.05$, $b = 0.2825$, $c = 0.65$, $d = 0.0175$
$t1 = a + b + c$, $t2 = a + b + d$, $x1 = t1 + d - 1$, $x2 = t2 + c - 1$
Comparaison de $x1$ et $x2$, Conclusion, explications ?

–b– Faire un programme pour calculer $a+n\ x$ de trois façons différentes :
–1– De cette manière
–2– Comme $((((a + x) + x) + x) + \cdots + x)$
–3– Comme $(a + \cdots + (x + (x + (x + x)))))$
On pourra faire des essais avec $a = 1$, $n = 1000000$ (pas plus !) et $x = 0.1$ ou $x = 0.01$ ou $x = 0.001$ ou $x = 10^{-7}$ ou $x = 10^{-8}$.

–c– Calculer, en sommant par le début et par la fin, la somme des inverses des carrés des N premiers entiers (non nuls).
En SP, prendre $N = 10000$ et $N = 100000$
En DP, prendre $N = 10^6$, $N = 10^7$ et $N = 10^8$ (pour les deux derniers cas, le faire qu'une seule fois ! !)
–1– Comparer avec le résultat exact.
–2– Avec un peu de maths, on peut améliorer le résultat numérique grâce au reste de la série. Le faire.

• 4 • Calcul d'un polynôme

Soit $P(x, y) = 2\ x^2 - x^4 + 9\ y^4$.

–a1– Calculer, avec des entiers, la valeur de ce polynôme aux points $(2, 1)$, $(7, 4)$, $(97, 56)$, $(18817, 10864)$

–a2– Calculer, avec des réels SP, la valeur de ce polynôme aux points $(2, 1)$, $(7, 4)$, $(97, 56)$, $(18817, 10864)$

–a3– Calculer, avec des réels DP, la valeur de ce polynôme aux points $(2, 1)$, $(7, 4)$, $(97, 56)$, $(18817, 10864)$

–b– Pour les courageux, relation de formation de ces (x_n, y_n) , démonstration et explications.

• 5 • Équations du second degré

Faire un programme pour résoudre l'équation du second degré :

$$ax^2 + bx + c = 0 \ .$$

On prendra pour (a, b, c) les triplets de valeurs suivantes :
$(12, -84, 147)$ $(1.2, -8.4, 14.7)$ $(0.12, -0.84, 1.47)$
Aussi : $(3, -21, 36.75)$ $(0.3, -2.1, 3.675)$.
Aussi : $(6, 5, -4)$ $(10^{-30}, -10^{30}, 10^{30})$.

–a– On le ferra à la main, en SP et en DP .

–b– On pourra aussi étudier l'équation : $x^2 - \lambda x + 1 = 0$ (prendre pour des applications numériques des valeurs $\lambda = 10$ ou $\lambda = 100$.

Du danger de l'utilisation des opérateurs << et >>

Sauf cas très spécifique (dûment justifié et donc commenté), il ne faut pas utiliser ces opérateurs (qualifiés d'abstrait) pour faire des manipulations arithmétiques : pour cela, il existe les opérateurs arithmétiques qui le font (en particulier, multiplication et division) !

Voici le programme écrit en C qui propose une astuce GEEEEENIALE pour faire les multiplications et divisions par 2 en faisant seulement des décalages de bits SAUF QUE …… cela ne se passe pas du tout comme prévu en particulier pour les nombres négatifs : voir les explications à la fin.

```
#include <stdio.h>

void main (void)
{
  signed   char      scI  , scJ  , scK  ;
  unsigned char      ucI  , ucK  , uc1  ;

  signed   short int ssiI , ssiJ , ssiK ;
  unsigned short int usiI , usiK , usi1 ;

  int i ;

  puts ( " Dans les exemples qui suivent, on CROIT avoir trouve une astuce " ) ;
  puts ( "     pour diviser et/ou multiplier rapidement par 2 MAIS .... " ) ;
  puts ( " " ) ;

  puts ( " d'abord avec des char : " ) ;

  uc1 = (unsigned char) 1 ;
  ucI = (unsigned char) 128 ;
  ucK = (unsigned char) 1 ;

  scI = (signed char) 64 ;
  scJ = (signed char) (-128) ;
  scK = (signed char) 1 ;

  for ( i = 0 ; i <= 9 ; ++i )
  {
    printf( " i = %1d , ucI = %4d , ucK = %4d , scI = %4d , scJ = %4d , scK = %4d \n",
            i , ucI , ucK , scI , scJ , scK ) ;
    ucI = ucI >> uc1 ;
    ucK = ucK << uc1 ;
    scI = scI >> uc1 ;
    scJ = scJ >> uc1 ;
    scK = scK << uc1 ;
  }
  puts ( " " ) ;
  puts ( " " ) ;


  puts ( " et apres des short int : " ) ;

  usi1 = (unsigned short int) 1 ;
  usiI = (unsigned short int) 32768 ;
  usiK = (unsigned short int) 1 ;
  ssiI = (signed short int) 16384 ;
  ssiJ = (signed short int) (-32768) ;
  ssiK = (signed short int) 1 ;
```

```
  for ( i = 0 ; i <= 17 ; ++i )
  {
    printf( " i = %2d , usiI = %6d , usiK = %6d , ssiI = %6d , ssiJ = %6d , ssiK = %6d \n",
            i , usiI , usiK , ssiI , ssiJ , ssiK ) ;
    usiI = usiI >> usi1 ;
    usiK = usiK << usi1 ;
    ssiI = ssiI >> usi1 ;
    ssiJ = ssiJ >> usi1 ;
    ssiK = ssiK << usi1 ;
  }
  puts ( " " ) ;
}
```

et voici les résultats ‘étonnants’ que cela donne :

```
Dans les exemples qui suivent, on CROIT avoir trouve une astuce
     pour diviser et/ou multiplier rapidement par 2 MAIS ....

 d'abord avec des char :
 i = 0 , ucI =  128 , ucK =    1 , scI =   64 , scJ = -128 , scK =    1
 i = 1 , ucI =   64 , ucK =    2 , scI =   32 , scJ =  -64 , scK =    2
 i = 2 , ucI =   32 , ucK =    4 , scI =   16 , scJ =  -32 , scK =    4
 i = 3 , ucI =   16 , ucK =    8 , scI =    8 , scJ =  -16 , scK =    8
 i = 4 , ucI =    8 , ucK =   16 , scI =    4 , scJ =   -8 , scK =   16
 i = 5 , ucI =    4 , ucK =   32 , scI =    2 , scJ =   -4 , scK =   32
 i = 6 , ucI =    2 , ucK =   64 , scI =    1 , scJ =   -2 , scK =   64
 i = 7 , ucI =    1 , ucK =  128 , scI =    0 , scJ =   -1 , scK = -128
 i = 8 , ucI =    0 , ucK =    0 , scI =    0 , scJ =   -1 , scK =    0
 i = 9 , ucI =    0 , ucK =    0 , scI =    0 , scJ =   -1 , scK =    0


 et apres des short int :
 i =  0 , usiI =  32768 , usiK =      1 , ssiI =  16384 , ssiJ = -32768 , ssiK =      1
 i =  1 , usiI =  16384 , usiK =      2 , ssiI =   8192 , ssiJ = -16384 , ssiK =      2
 i =  2 , usiI =   8192 , usiK =      4 , ssiI =   4096 , ssiJ =  -8192 , ssiK =      4
 i =  3 , usiI =   4096 , usiK =      8 , ssiI =   2048 , ssiJ =  -4096 , ssiK =      8
 i =  4 , usiI =   2048 , usiK =     16 , ssiI =   1024 , ssiJ =  -2048 , ssiK =     16
 i =  5 , usiI =   1024 , usiK =     32 , ssiI =    512 , ssiJ =  -1024 , ssiK =     32
 i =  6 , usiI =    512 , usiK =     64 , ssiI =    256 , ssiJ =   -512 , ssiK =     64
 i =  7 , usiI =    256 , usiK =    128 , ssiI =    128 , ssiJ =   -256 , ssiK =    128
 i =  8 , usiI =    128 , usiK =    256 , ssiI =     64 , ssiJ =   -128 , ssiK =    256
 i =  9 , usiI =     64 , usiK =    512 , ssiI =     32 , ssiJ =    -64 , ssiK =    512
 i = 10 , usiI =     32 , usiK =   1024 , ssiI =     16 , ssiJ =    -32 , ssiK =   1024
 i = 11 , usiI =     16 , usiK =   2048 , ssiI =      8 , ssiJ =    -16 , ssiK =   2048
 i = 12 , usiI =      8 , usiK =   4096 , ssiI =      4 , ssiJ =     -8 , ssiK =   4096
 i = 13 , usiI =      4 , usiK =   8192 , ssiI =      2 , ssiJ =     -4 , ssiK =   8192
 i = 14 , usiI =      2 , usiK =  16384 , ssiI =      1 , ssiJ =     -2 , ssiK =  16384
 i = 15 , usiI =      1 , usiK =  32768 , ssiI =      0 , ssiJ =     -1 , ssiK = -32768
 i = 16 , usiI =      0 , usiK =      0 , ssiI =      0 , ssiJ =     -1 , ssiK =      0
 i = 17 , usiI =      0 , usiK =      0 , ssiI =      0 , ssiJ =     -1 , ssiK =      0
```

- Pour des ‘*unsigned integer*’ ou des ‘*signed integer*’ positifs, la division par 2 obtenue à l’aide d’un décalage de bits donne bien le comportement attendu (rappel : 1/2 donne bien 0 en division entière).
- Pour des ‘*signed integer*’ négatifs, la division par 2 obtenue à l’aide d’un décalage de bits ne donne plus du tout le comportement attendu (‘-1’ est l’entier composé que de bits égaux à un et est donc insensible à tout décalage de bits vers la droite).
- Pour des ‘*unsigned integer*’ positifs, la multiplication par 2 obtenue à l’aide d’un décalage de bits donne bien le comportement attendu (rappel : en ‘C’, la notion d’ ‘*integer overflow*’ n’existe pas, donc, après la plus grande puissance de 2 représentable, un ‘un’ suivi que de zéros, vient bien un nombre composé que de bits à zéros et donc égal à 0).
- Pour des ‘*signed integer*’ positifs, la multiplication par 2 obtenue à l’aide d’un décalage de bits ne donne plus du tout le comportement attendu (rappel : en ‘C’, la notion d’ ‘*integer overflow*’ n’existe pas ; après la plus grande puissance (positive) de 2 représentable, le nombre obtenu, comportant un bit égal à un en première position, est interprété comme nombre négatif ; la valeur ainsi obtenue est donc (pour des ‘*char*’) équivalente à : 64 + 64 , soit la même chose que 127 +1 , soit bien – 128 (*cf.* cours).

IEEE Standard 754 Floating Point Numbers

From Steve Hollasch, Modified (!) by Bernard Pichon / Last update 2003-Jan-02

Storage Layout

IEEE floating point numbers have three basic components: the sign, the exponent, and the mantissa. The mantissa is composed of the *fraction* and an implicit leading digit (explained below). The exponent base (2) is implicit and need not be stored.

The following figure shows the layout for single (32-bit) and double (64-bit) precision floating-point values. The number of bits for each field are shown (bit ranges are in square brackets):

	Sign	Exponent	Fraction	Bias
Single Precision	1 [31]	8 [30-23]	23 [22-00]	127
Double Precision	1 [63]	11 [62-52]	52 [51-00]	1023

The Sign Bit

The sign bit is as simple as it gets. 0 denotes a positive number; 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.

The Exponent

The exponent field needs to represent both positive and negative exponents. To do this, a *bias* is added to the actual exponent in order to get the stored exponent.

For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73. For reasons discussed later, exponents of -127 (all 0s) and +128 (all 1s) are reserved for special numbers.

For double precision, the exponent field is 11 bits, and has a bias of 1023.

The Mantissa

The *mantissa*, also known as the *significand*, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits.

A nice little optimization is available to us in base two, since the only possible non-zero digit is 1. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by way of 23 fraction bits.

Ranges of Floating-Point Numbers

The range of positive floating point numbers can be split into normalized numbers (which preserve the full precision of the mantissa), and *denormalized* numbers (discussed later) which use only a portion of the fractions's precision.

	Denormalized	Normalized	Approximate Decimal
Single Precision	$\pm 2^{-149}$ to $(1-2^{-23})x2^{-126}$	$\pm 2^{-126}$ to $(2-2^{-23})x2^{127}$	$\pm \sim 10^{-44.85}$ to $\sim 10^{38.53}$
Double Precision	$\pm 2^{-1074}$ to $(1-2^{-52})x2^{-1022}$	$\pm 2^{-1022}$ to $(2-2^{-52})x2^{1023}$	$\pm \sim 10^{-323.3}$ to $\sim 10^{308.3}$

Since the sign of floating point numbers is given by a special leading bit, the range for negative numbers is given by the negation of the above values.

Special Values

IEEE reserves exponent field values of all 0s and all 1s to denote special values in the floating-point scheme.

Zero

As mentioned above, zero is not directly representable in the straight format, due to the assumption of a leading 1 (we'd need to specify a true zero mantissa to yield a value of zero). Zero is a special value denoted with an exponent field of zero and a fraction field of zero. Note that -0 and +0 are distinct values, though they both compare as equal.

Denormalized

If the exponent is all 0s, but the fraction is non-zero (else it would be interpreted as zero), then the value is a *denormalized* number, which does *not* have an assumed leading 1 before the binary point. Thus, this represents a number $(-1)^s \times 0.f \times 2^{-126}$, where s is the sign bit and f is the fraction. For double precision, denormalized numbers are of the form $(-1)^s \times 0.f \times 2^{-1022}$. From this you can interpret zero as a special type of denormalized number.

Infinity

The values +infinity and -infinity are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations. *Operations with infinite values are well defined in IEEE floating point.*

Not A Number

The value NaN (*Not a Number*) is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1s and a non-zero fraction. There are two categories of NaN: QNaN (*Quiet NaN*) and SNaN (*Signalling NaN*).

A QNaN is a NaN with the most significant fraction bit set. QNaN's propagate freely through most arithmetic operations. These values pop out of an operation when the result is not mathematically defined.

An SNaN is a NaN with the most significant fraction bit clear. It is used to signal an exception when used in operations. SNaN's can be handy to assign to uninitialized variables to trap premature usage.

Semantically, QNaN's denote *indeterminate* operations, while SNaN's denote *invalid* operations.

Summary (see last page)

References

A lot of this stuff was observed from small programs I wrote to go back and forth between hex and floating point (*printf*-style), and to examine the results of various operations. The bulk of this material, however, was lifted from Stallings' book.

1. *Computer Organization and Architecture*, William Stallings, pp. 222 -234 Macmillan Publishing Company, ISBN 0 -02-415480-6

2. IEEE Computer Society (1985), *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE Std 754-1985.

3. *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, (a PDF document from Intel Web site.)

See Also

?? IEEE Standards Site (http://standards.ieee.org/)

Summary

To sum up, the following are the corresponding values for a given representation:

Float Values (b = bias)			
Sign	**Exponent (e)**	**Fraction (f)**	**Value**
0	00..00	00..00	+0
0	00..00	00..01 : 11..11	Positive Denormalized Real $0.f \times 2^{(-b+1)}$
0	00..01 : 11..10	XX..XX	Positive Normalized Real $1.f \times 2^{(e-b)}$
0	11..11	00..00	+Infinity
0	11..11	00..01 : 01..11	SNaN
0	11..11	10..00 : 11..11	QNaN
1	00..00	00..00	-0
1	00..00	00..01 : 11..11	Negative Denormalized Real $-0.f \times 2^{(-b+1)}$
1	00..01 : 11..10	XX..XX	Negative Normalized Real $-1.f \times 2^{(e-b)}$
1	11..11	00..00	-Infinity
1	11..11	00..01 : 01..11	SNaN
1	11..11	10..00 : 11.11	QNaN