◊ Ecrivez la fonction moyenne avec le prototype suivant :

`float moy(float somme , int N);`

L'objectif est ici clairement de manipuler les fonctions. Un rappel sur leur utilisation est ici.

Solution :

```c
#include<stdio.h>

float moy( float somme, int n);

float moy( float somme, int n)
{    return somme/n;
}

int main(void)
{    float x[10];
 int k, N;
 float S;

 printf("Entrez le nombre de valeurs à saisir : ");
 scanf("%d",&N);

 for ( k=0 ; k<N; k++)
 {   printf("Entrez X(%d) : ",k);
     scanf("%f", &x[k]);
 }

 S = 0 ;
 for ( k=0 ; k<N ; k++)
     S = S + x[k];

 printf("La moyenne est %f\n",  moy(S,N) );

 return 0;
}
```

Remarques sur le programme :

Ce programme crée une fonction qui ne fait qu'un seul calcul : somme/n. Le calcul est d'ailleurs effectué dans l'instruction return. Notez que le résultat du calcul doit impérativement être du même type que celui indiqué lors de la déclaration de la fonction par son prototype (ligne 3).

Cette fonction a deux paramètres : somme et n. Elle retourne un float et il est utilisé directement dans le printf ligne 26. Le compilateur va générer l'appel de la fonction moy en initialisant ses deux paramètres somme et n respectivement avec les valeurs contenues dans S et N. Les valeurs étant copiées dans somme et n, S et N ne sont pas modifiées par la fonction.

On peut envisager de réaliser le calcul de la somme dans la fonction moy directement :

```
 1 #include<stdio.h>
 2
 3 float moy( float X[10], int N);
 4
 5 float moy( float X[10], int N)
 6 {    int k;
 7  float somme;
 8
 9  somme = 0;
10  for ( k=0 ; k<N ; k++ )
11      somme = somme + X[k];
12
13  return somme/N;
14 }
15
16 int main(void)
17 {    float x[10];
18  int k, n;
19
20  printf("Entrez le nombre de valeurs à saisir : ");
21  scanf("%d",&n);
```

```
22
23  for ( k=0 ; k<n; k++)
24  {   printf("Entrez X(%d) : ",k);
25      scanf("%f", &x[k]);
26  }
27
28  printf("La moyenne est %f\n",  moy(x,n) );
29
30  return 0;
31 }
```

Remarques :

ligne 6 : k est une variable locale à la fonction moy(), elle est totalement indépendante de la variable k définie ligne 18.

Pas de grosses surprises dans ce programme, toutefois, on peut l'améliorer en utilisant des types de façon à le rendre plus lisible.

```
 1 #include<stdio.h>
 2
 3 typedef float vect[10] ;
 4
 5 float moy( vect X, int N);
 6
 7 float moy( vect X, int N)
 8 {   int k;
 9  float somme;
10
11  somme = 0;
12  for ( k=0 ; k<N ; k++ )
13      somme = somme + X[k];
14
15  return somme/N;
16 }
17
18 int main(void)
19 {   vect x;
```

```
20  int k, n;
…
```

L'utilisation de type simplifie le programme. Nous verrons, au semestre prochain, que cette utilisation permet un certain contrôle des dimensions des variables dimensionnées dans le programme. En effet le compilateur attend une variable de type vect pour la fonction moy, il n'est pas possible de lui envoyer des variables dimensionnées de taille trop courte (ou trop longue). De plus il est simple de re-dimensionner toutes les variables, il suffit de modifier le type en début de programme. Aucun risque de se tromper.

◊ Ecrivez la fonction fact( ) avec le prototype suivant :
`float fact( int N );`

Solution :

Il y a plusieurs façons de procéder, la plus efficace est la suivante :

```
 1 #include<stdio.h>
 2
 3 float fact( int N) ;
 4
 5 float fact( int N)
 6 {   float F;
 7  int k ;
 8
 9  F = 1 ;
10  for ( k=1 ; k<=N ; k++)
11      F *= k;
12  return F;
13 }
14
15 int main(void)
16 {   int  n;
```

```
17
18  printf("Entrez la valeur de n : ");
19  scanf("%d", &n  ) ;
20
21  printf("%d! = %g \n", n, fact(n) );
22
23  return 0;
24 }
```

Commentaires :

Pas de surprises ici. On a juste créé une fonction et on y a copié et adapté le code du programme de l'exercice 4.
Notez l'utilisation de l'opérateur combiné ligne 11 : F *= k; équivaut à F = F*k;

Cette méthode est dite séquentielle, il est possible d'envisager une autre méthode, dite récursive.

```
 1 #include<stdio.h>
 2
 3 float fact( int N) ;
 4
 5 float fact( int N)
 6 {    float F;
 7
 8  if (N>1)
 9      F = N*fact(N-1);
10  else
11      F = 1;
12
13  return F;
14 }
15
16 int main(void)
17 {    int  n;
18
```

```
19  printf("Entrez la valeur de n : ");
20  scanf("%d", &n  ) ;
21
22  printf("%d! = %g \n", n, fact(n) );
23
24  return 0;
25 }
```

Comment ce programme fonctionne-t-il ?

L'idée de départ est simple : N! = N*(N-1) ! = N*(N-1)*(N-2) ! = N*(N-1)*…*2*1! = N*(N-1)*…*2*1

Donc pour calculer N!, il suffit de multiplier N par (N-1)!. Mais n'avons-nous pas une fonction qui calcule (N-1)! ? Clairement si : la fonction fact ! Par conséquent, nous allons calculer N! en multipliant par N fact(N-1).

En fait la fonction s'appelle elle-même. Il doit bien sur y avoir un moment ou l'on stoppe le processus, c'est lorsque la valeur de la factorielle est évidente : 0! = 1! = 1. A ce moment là le calcul est fini.

Essayez de faire à la main le calcul de 4! par cette méthode...

Elle est intellectuellement très intéressante, mais peu efficace du point de vue informatique dans le cas présent (c'est le mauvais cas d'école, parce qu'il y a des problèmes que l'on ne peut quasiment résoudre qu'avec une approche de ce type).

En quoi cette approche est-elle mauvaise ici ? Tout simplement parce qu'il existe une approche directe qui fonctionne fort bien !
Nous créons une fonction qui s'appelle elle même, et dont le résultat ne sera calculé qu'une fois qu'elle s'appellera avec comme valeur 1. Le système doit donc gérer N-1 appels de fonctions avec

retour au point de départ....

Que fait le système quand il appelle une fonction ?

En fait lors de l'appel d'une fonction, le système doit impérativement savoir ou revenir pour reprendre le programme. Il stocke les variables à transmettre, le contexte actuel (contenu de la mémoire) et exécute la fonction, récupère le résultat et restaure le contexte d'origine.

L'appel de fonction est donc une opération lourde qui sollicite beaucoup le processeur et surtout consomme pas mal de mémoire. Imaginez la mémoire utilisée pour calculer 1500 !

La boucle elle, ne procède qu'à un appel.

Au final, les algorithmes récursifs sont à éviter à moins de n'avoir aucune autre solution algorithmique.