Ecriture, lecture, fichiers

Préambule : rudiments pratiques

On indique ici le strict minimum pour pouvoir lire des données et écrire des résultats de la manière la plus élémentaire, de façon à pouvoir écrire des programmes avant d'avoir lu l'ensemble du chapitre.

Ecrire sur l'écran

Pour écrire la valeur d'une variable :

- `i` de type `int` : `printf("%d\n",i);`
- `x` de type `double` : `printf("%lg\n",x);`
- `k` de type `char` : `printf("%c\n",k);`

Pour écrire des commentaires, par exemple : *Valeur de i :*

```
printf("Valeur de i\n");
```

Pour écrire les valeurs de `i`, `x` et `k` avec des commentaires :

```
printf("Valeur de i : %d Valeur de x : %lg Valeur de k : %c\n",i,x,k);
```

Lire sur le clavier

Pour lire la valeur d'une variable :

- `i` de type `int` : `scanf("%d",&i);` (prendre garde à l'esperluète précédent le `i`)
- `x` de type `double` : `scanf("%lg",&x);`
- `k` de type `char` : `scanf(" %c",&k);` (prendre garde au blanc précédent `%c`)

Lors de l'exécution de la lecture la valeur frappée au clavier doit être validée par un *Return*[1].

Pour lire tout à la fois :

Lire les valeurs de `i`, `x` et `k` en une seule fois :

```
scanf("%d%lg %c",&i,&x,&k);
```

Lors de l'exécution de la lecture, les valeurs frappées au clavier doivent être séparées par des blancs ou des *Return* et la dernière (celle de `k` ici), suivie d'un *Return* de validation.

Vif du sujet

1 Ecriture de résultats à l'écran

1.1 Base de la méthode

L'instruction d'écriture à l'écran se présente selon le schéma :

Ecrire : comment, quoi

plus précisément :

`printf("`*format*`",`*expressions*`)`

qui indique que les valeurs des *expressions* seront écrites suivants les spécifications contenues dans la chaîne de caractères *format*. Les expressions peuvent bien sûr se réduire à une variable ou une constante explicite ou non. Un format est composé de « descripteurs » , débutant toujours par le caractère `%`, et de texte. A chaque descripteur correspond une expression, la correspondance se faisant dans l'ordre :

```
printf("%desc1 %desc2 %desc3 ...",expr1,expr2,expr3,...)
```

Sous leur forme la plus simple les formats peuvent s'écrire à l'aide des descripteurs suivants :

[1] sur certains claviers la touche *Return* se nomme *Enter*

`%d` pour le type `int`
`%lg` pour le type `double`
`%c` pour le type `char`

```
...
int k=1;double x=2.;char c='a';
...
printf("%d %lg %c\n",k,x,c);
...
```

donne le résultat :
1 2 a
Le format peut contenir en plus : des commentaires, des blancs, des passages à la ligne. Par exemple :

```
printf("Valeur de k : %d   Valeur de x : %lg   Valeur de c : %c\n",k,x,c);
```

donne le résultat :
Valeur de k : 1 Valeur de x : 2 Valeur de c : a
Tandis que :

```
printf("Valeur de k : %d\nValeur de x : %lg\nValeur de c : %c\n",k,x,c);
```

donne :
Valeur de k : 1
Valeur de x : 2
Valeur de c : a

1.2 Paramétrage des descripteurs

Les descripteurs `%d`, `%lg`, `%c` peuvent être précisés par des paramètres optionnels.
De façon générale un descripteur se présente sous la forme :

%justif taille .precis long type

où *type* est seul obligatoire.

type est le type de la variable (ou constante, constante explicite ou expression) à imprimer :

- `d` pour les entiers
- `e`, `f` ou `g` pour les réels :
  - `e` pour une notation exponentielle
  - `f` pour une notation en point décimal fixe
  - `g` pour la notation la plus lisible des deux précédentes, le choix étant fait automatiquement par la machine
- `c` pour les caractères

long est :

- absent pour les entiers de type `int`[1], les réels de type `float`[1] et les caractères [2]
- égal à `l` pour les entiers de type `long int`[1] et les réels `double`[3]

taille indique le nombre minimal de colonnes[4] sur lesquelles on demande l'écriture de la valeur (y compris signe, point décimal et exposant quand il y en a). Si la taille est insuffisante elle est augmentée automatiquement de ce qu'il faut.
Par défaut la justification est à droite. Si l'on veut compléter à gauche par des 0 et non par des blancs on ajoute un 0 devant le paramètre taille.

.precis ne concerne que les réels, c'est le nombre souhaité de chiffres après le point décimal ou le nombre de chiffres significatifs, selon qu'on utilise `e`, `f` ou `g`.

justif vaut `-` pour une justification à gauche et est absent dans le cas contraire.

[1] 4 octets sur les PC utilisés
[2] 1 octet sur les PC utilisés
[3] 8 octets sur les PC utilisés
[4] une colonne=un caractère

1.3 Interprétation du format

Le format doit être regardé comme un petit « programme » auquel on fournit des données et qui exécute une certaine tâche. Par exemple dans l'instruction :

```
printf("Valeur de n : %d Valeur de x : %lg\n",n,x)
```

ce qui est entre `"..."` est le programme, et `n` et `x` sont les données.
Le `printf` demande l'exécution pas à pas des tâches suivantes :

écrire *Valeur de n :*
écrire un blanc
écrire la valeur de `n` selon le descripteur `%d`
écrire un blanc
écrire *Valeur de x :*
écrire un blanc
écrire la valeur de `x` selon le descripteur `%lg`
se positionner à la ligne suivante.

Cette façon de voir permet de comprendre certains comportements indiqués au paragraphe suivant.

1.4 Non respect des règles

1.4.1 Non égalité du nombre de descripteurs et du nombre d'expressions

- S'il y a plus d'expressions à imprimer que de descripteurs les expressions en trop sont ignorées : le « programme » « format » ne lit pas toutes ses données. Elles sont définitivement ignorées car elles ne seront pas reprises par un éventuel format ultérieur. Il n'y a donc rien de faux mais il manque des résultats.
- S'il y a plus de descripteurs que d'expressions le « programme » « format » lit les données manquantes n'importe où, les conséquences sont incontrôlables et les résultats éventuels totalement aberrants.

1.4.2 Non respect du type du descripteur avec celui de l'expression à imprimer

Si le type du descripteur et celui de l'expression sont différents :
- le résultat est aberrant
- les valeurs des expressions suivantes éventuelles à imprimer dans le même format également.

Ce dernier point est vrai même si le type du descripteur et celui de l'expression, bien que différents, correspondent, au moins apparemment, au même nombre d'octets. Par exemple :

```
...
int i; float x; double y;
i=1; x=2.; y=3.;
printf("%d %g\n",i,x);
printf("%d %d\n",i,x);
printf("%g %g\n",i,x);
printf("%d %lg\n",i,y);
printf("%d %d\n",i,y);
printf("%lg %lg\n",i,y);
return (0);
...
```

donne le résultat :
1 2
1 0
4.94066E-324 -1.99843
1 3
1 0
4.94066E-324 -1.99843

2 Entrée de données depuis le clavier

L'instruction permettant d'attribuer une valeur à une variable à partir du clavier, et qui réalise donc l'opération inverse de `printf`, est `scanf`. Ces deux instructions présentent des similitudes mais aussi des différences importantes.

2.1 Exemple

Pour entrer un nombre entier dans une variable de type `int` à partir du clavier on écrit par exemple :

```
...
int i;
...
scanf("%d",&i);
...
```

`scanf` comme `printf`, utilise des descripteurs de format, `%d` dans cet exemple, et ce sont les mêmes. Mais on remarque que le nom de la variable[2] est précédé d'une esperluète. On verra au chapitre sur les pointeurs que `&i` désigne l'« adresse » en mémoire de la variable `i` et non la variable elle-même. La raison pour laquelle c'est l'adresse qu'il faut indiquer apparaîtra dans ce même chapitre.
Lorsqu'il arrive à l'instruction `scanf` le programme s'arrête et attend que le clavier lui fournisse une suite de caractères. L'utilisateur frappe ces caractères (qui s'inscrivent aussi à l'écran) en terminant par un *Return*. Par exemple :

123

Pour le programme ce *Return* est le signal de départ de la lecture par `scanf`. Puisque `scanf` contient un descripteur de format `%d` le programme cherche à interpréter la suite de caractères comme un entier. S'il y parvient il place cette valeur dans la variable `i` et c'est terminé.

De même, pour entrer un nombre réel dans une variable de type `double` :

```
...
double x;
...
scanf("%lg",&x);
...
```

et pour entrer une valeur caractère dans une variable de type `char` :

```
...
char k;
...
scanf(" %c",&k);
...
```

2.2 Mécanisme de l'entrée des données

Pour bien comprendre comment s'exécute le `scanf` il faut examiner la façon dont les caractères[3] sont transmis du clavier au programme en train de s'exécuter. Supposons que l'on tape au clavier la suite de caractères :

vc.d1␣;g5n,6

où ␣ symbolise un blanc, qui est un caractère comme un autre[4]. Chaque caractère est inscrit dans un fichier intermédiaire dit « tampon » (« buffer » en anglais). L'état du tampon peut alors être représenté de la façon suivante :

vc.d1␣;g5n,6\n

où \n symbolise *Return* qui est aussi considéré comme un caractère et qui est, lui aussi, écrit dans le tampon. La frappe de *Return* provoque la lecture du tampon par `scanf`. On a vu en étudiant `printf` que le format s'interprète comme un petit programme qui lit des données en entrée selon certains descripteurs et les écrit sur une sortie. Ici c'est l'inverse du `printf` : les données viennent de l'extérieur du programme et sont écrites dans des variables.

2.3 Cas des nombres entiers ou réels

Ce qui suit va être illustré dans le cas des entiers mais s'applique identiquement aux réels.
Supposons que le format du `scanf` soit, par exemple :

```
...
int i,j;
...
scanf("%d%d",&i,&j)
...
```

[2] bien sûr, contrairement au cas de `printf`, il ne peut s'agir que d'une variable et non pas d'une expression

[3] à ce stade il n'y a que des caractères, même si certains de ces caractères peuvent être des chiffres. Ce n'est qu'au moment de l'exécution du `scanf` qu'éventuellement ces caractères pourront être interprétés comme des nombres

[4] dans la suite on suppose toujours que chaque ligne frappée au clavier est terminée par un *Return*

et que l'on entre au clavier :

123␣456

L'état initial du tampon est alors :

123␣456\n

Le premier `%d` cherche à lire un entier en analysant le tampon caractère par caractère : il commence par sauter tous les blancs et \n initiaux éventuellement présents (dans cet exemple il n'y en a pas), puis lit tous les caractères qu'il rencontre jusqu'à trouver un blanc ou un \n (qui jouent donc le rôle de « séparateurs » [5]).

2.3.1 Premier cas : le %d parvient à interpréter ce qu'il lit comme un entier

Il met alors la valeur dans la variable `i`, et efface du tampon ce qu'il vient de lire, non compris le séparateur sur lequel il s'est arrêté. L'état du tampon est alors :

␣456\n

Ensuite c'est le second `%d` qui s'exécute en poursuivant la lecture du tampon : il saute le blanc, lit les caractères jusqu'au \n et les interprète comme une valeur entière qu'il met dans la variable `j`. L'état du tampon est alors :

\n

c'est à dire, c'est important, qu'il n'est pas vide.
Dans l'exemple précédent tout se passe correctement et à la fin `i` vaut `123` et `j` `456`.
Si l'on avait entré au clavier :

123
456

c'est à dire en séparant les deux nombres par un *Return* et non par un blanc, on voit, en appliquant le mécanisme qui vient d'être décrit, que le résultat serait identique.

Bien entendu si l'on avait entré au clavier :

123␣45␣6

c'est à dire un blanc supplémentaire entre *5* et *6*, `j` vaudrait `45` à la fin et l'état du tampon après le `scanf` serait :

␣6\n

Le fait que le tampon ne soit pas vidé systématiquement après le `scanf` a des conséquences visibles dans l'exemple du programme :

```
#include<stdio.h>
int main(void)
{
int i,j,k;
scanf("%d%d",&i,&j);
printf("%d %d\n",i,j);
scanf("%d",&k);
printf("%d\n",k);
return (0);
}
```

auquel on entre au clavier :

123␣456
789

et qui donne le résultat attendu :

123␣456
789

tandis que si on entre :

123␣45␣6
789

le résultat est :

123␣45
6

[5] une liste de séparateurs analogues à \n est donnée à l'annexe B

et il reste dans le tampon :

*n789**n*

ce qui signifie que tous les `scanf` suivants seront en retard d'une donnée.
Il faut donc retenir le fait important : le tampon n'est pas systématiquement vidé après chaque `scanf`

2.3.2 Second cas : le `%d` ne parvient pas interpréter ce qu'il lit comme un entier

Le `scanf`, dans sa totalité, c'est à dire y compris les descripteurs non encore utilisés, s'arrête alors, en laissant tout ce qu'il n'a pu interpréter tel quel dans le tampon, mais le programme se poursuit et, évidemment, les éventuels `scanf` suivants seront affectés.

2.3.3 Utilisation de descripteurs détaillés

Au lieu d'un simple `%d` ou `%lg` les descripteurs peuvent être complétés par des paramètres comme dans le cas de `printf`, bien que cela soit surtout utile lorsqu'on lit dans un fichier plutôt que sur le clavier.
Pour des entiers on peut par exemple écrire :

```
...
int i,j;
...
scanf("%3d%3d",i,j);
...
```

et le `scanf` cherche à interpréter comme un entier ce qu'il trouve dans les trois premières colonnes maximum, ou dans les trois colonnes maximum suivant un blanc ou un *Return* et non pas ce qu'il trouve jusqu'au blanc ou *Return* suivant. On peut alors entrer au clavier :

123␣456

ou

123456

dans ces deux cas le résultat sera correct.
De même avec des réels on peut écrire :

```
...
int x,y;
...
scanf("%5.2f%8.3f",x,y);
...
```

et `scanf` limite sa lecture de `x` à cinq colonnes maximum et celle de `y` à huit maximum.

2.4 Cas des caractères

Il y a une seule différence avec le cas des nombres entiers ou réels mais elle est importante : le descripteur `%c` ne saute ni blanc ni *n*. Cela explique certains résultats inattendus, par exemple avec le programme suivant :

```
#include<stdio.h>
int main(void)
{
int i; char k;
scanf("%d%c",&i,&k);
printf("%d%c\n",i,k);
return (0);
}
```

Si on tape au clavier :

1␣a

le tampon est :

*1␣a**n*

et on obtient à l'écran :

1␣

Si on tape au clavier :

1

le tampon est :

1$\backslash n$

et on obtient à l'écran :

1

␣

C'est seulement si on tape :

1a

qui donne le tampon :

1a$\backslash n$

que l'on obtient à l'écran le résultat attendu :

1a

Pour y remédier on peut écrire :

```
scanf("%d %c",&i,&k);
```

c'est à dire ajouter un blanc avant le `%c` (voir section suivante).
De même :

```
/* Calcul en boucle du carre d'un nombre */
#include<stdio.h>
int main(void)
{
  int n; char rep;
do
   {
   printf("n=? "); scanf("%d",&n);
   printf("%d au carré=%d\n",n,n*n);
   printf("Voulez-vous continuer ? y/[n] "); scanf("%c",&rep);
   if(rep!='y') break;
   }
while(1);
return (0);
}
```

sort de la boucle après le premier tour sans attendre la réponse car `%c` met dans `rep` le $\backslash n$ resté dans le tampon après la lecture de `n`, ce qui donne une réponse différente de `y`.
Pour y remédier on peut remplacer `scanf("%c",&rep);` :
- soit par `scanf(" %c",&rep);` on ajoute un blanc avant le `%c`, qui absorbe le $\backslash n$ (voir section suivante)
- soit par `scanf("%c%c",&rep,&rep);` le premier `%c` absorbe le $\backslash n$ (peu élégant)
- soit par `scanf("%1s",&rep);` on emploie un descripteur de chaîne de un caractère au lieu d'un descripteur de caractère[6]

la première méthode étant la plus simple.

2.5 Rôle d'éventuels caractères de texte entre les descripteurs

Jusqu'ici on a considéré des formats contenant uniquement des descripteurs. Comme dans `printf`, il est possible d'ajouter du texte avant, entre ou après les descripteurs :
- un blanc demande au format de consommer le maximum de blancs ou de $\backslash n$ dans le tampon (intérêt uniquement dans le cas des variables caractère)
- tout autre caractère demande au format d'identifier immédiatement le même caractère dans le tampon, sinon le `scanf` s'arrête.

Exemple 1[7] :

on veut sauter des blancs situés entre des variables caractère :

[6] les chaînes de caractères seront vues dans un chapitre ultérieur
[7] à sauter en première lecture

```
#include<stdio.h>
int main(void)
{
  char i,j,k;
  scanf(" %c %c %c",&i,&j,&k);
  printf("%c%c%c\n",i,j,k);
  scanf(" %c%c%c",&i,&j,&k);
  printf("%c%c%c\n",i,j,k);
  return (0);
}
```

On obtient :

clavier : *abc*
écran : *abc*
clavier : *abc*
écran : *abc*

mais :

clavier : *a␣␣␣b␣c*
écran : *abc*
clavier : *a␣␣␣b␣c*
écran : *a*

Exemple 2 :

on veut obliger l'utilisateur à entrer des dates sous la forme 7/10/2001 c'est à dire en séparant jour, mois, année par des /. On peut écrire :

```
#include<stdio.h>
int main(void)
{
int i,j,k;
scanf("%d/%d/%d",&i,&j,&k);
printf("%d/%d/%d\n",i,j,k);
return (0);
}
```

Remarque

attention, un blanc placé à la fin du format d'un `scanf` provoque un blocage de la lecture. Par exemple, il ne faut pas laisser de blanc après le `%d` suivant :

```
scanf("%d ",&i);
```

mais bien écrire :

```
scanf("%d",&i);
```

3 Valeurs de `printf` et `scanf`

On a vu que les fonctions `printf` et `scanf` effectuent certaines tâches. De plus elles renvoient une valeur[8] :

pour `printf` c'est le nombre de caractères imprimés
pour `scanf` c'est le nombre de valeurs lues avec succès

Si la tâche ne s'effectue pas normalement `printf` renvoit une valeur négative et `scanf` une valeur nulle ou la valeur contenue dans la variable prédéfinie qui se nomme `EOF`. Ceci permet de vérifier que `printf` ou `scanf` se sont bien passés et sinon d'interrompre éventuellement le déroulement du programme. En effet on a vu qu'après un `printf` ou un `scanf` défectueux, l'exécution se poursuit, bien que dans la plupart des cas cela n'ait plus aucun sens.

Exemple

```
#include<stdio.h>
int main(void)
{
int i,r;
r=scanf("%d",&i); printf("Nb de valeurs lues par scanf : %d\n",r);
r=printf("%d\n",i); printf("Nb de caractères écrits par printf : %d\n",r);
```

[8] ce sont des fonctions, voir le chapitre **Fonctions**

```
return (0);
}
```

Résultat :

clavier : *123*
écran : *Nb de valeurs lues par scanf : 1*
123
Nb de caractères écrits par printf : 4

Remarque[9]

On pourrait écrire le programme précédent de façon plus concise :

```
#include<stdio.h>
int main(void)
{
int i;
printf("Nb de valeurs lues par scanf : %d\n",scanf("%d",&i));
printf("Nb de caractères écrits par printf : %d\n",printf("%d\n",i));
return (0);
}
```

4 Fichiers

Au lieu d'imprimer des résultats à l'écran ou de lire des données à partir du clavier on peut réaliser ces opérations dans des fichiers, ce qui, pour nos besoins, n'entraîne que de petites modifications. Dans cette section on va indiquer une méthode très rudimentaire d'utilisation des fichiers. Nous ne considérerons que des fichiers ASCII, en accès séquentiel, ce qui suffit dans de nombreux cas.

4.1 Exemple d'écriture dans un fichier

On veut écrire la valeur d'une variable `i` dans un fichier dont le nom Linux sera *mon_prog.res* par exemple. On aura toujours le schéma suivant :

```
...
int i;
FILE *fich;
...
fich=fopen("mon_prog.res","w");
...
i=...
fprintf(fich,"%d\n",i);
...
fclose(fich);
...
```

Commentaire détaillé

`FILE *fich;` déclare une variable d'un type spécial[10] qu'on a choisi ici de nommer `fich` et qui est destinée à recevoir la désignation du fichier dans le programme C, différente de la désignation Linux qui est *mon_prog.res.*

`fich=fopen("mon_prog.res","w");` ouvre un fichier de nom *mon_prog.res* en Linux, le `"w"` signifiant que l'on veut écrire dans ce fichier. La désignation de ce fichier dans le programme C est placée dans la variable `fich`[11].

`fprintf(fich,"%d\n",i);` écrit la valeur de `i` dans le fichier *mon_prog.res* avec le format `%d\n`.

`fclose(fich);` referme le fichier *mon_prog.res* une fois qu'on n'a plus rien à écrire dedans.

L'exemple suivant est une illustration de ce schéma. Il écrit dans le fichier nommé *carres.res* en Linux, les carrés des `n` premiers entiers à partir de 1 :

```
/* Ecriture du fichier carres.res */
#include<stdio.h>
int main(void)
```

[9] à sauter en première lecture
[10] sa signification apparaîtra au chapitre **Pointeurs**
[11] donc elle n'apparait pas explicitement

```
{
int i;
FILE *fich;
fich=fopen("carres.res","w");
for(i=1; i<=n; i++) fprintf(fich,"%d\n",i*i);
fclose(fich);
}
```

4.2 Lecture dans un fichier

Le schéma est presqu'identique à celui de la section précédente :

```
...
int i;
FILE *fich;
...
fich=fopen("mon_prog.res","r");
...
i=...
fscanf(fich,"%d",&i);
...
fclose(fich);
...
```

on a seulement remplacé `"w"` par `"r"` dans le `fopen` et `fprintf` par `fscanf` puisqu'il s'agit d'une lecture au lieu d'une écriture.

Remarque

`fprintf` et `fscanf` s'utilisent exactement comme `printf` et `scanf`, il y a seulement un argument en plus servant à désigner le fichier dans lequel on veut lire ou écrire.
On peut d'ailleurs utiliser :

`fprintf(stdio,"%d\n",i)` au lieu de `printf("%d\n",i)` pour écrire à l'écran
`fscanf(stdio,"%d",&i)` au lieu de `scanf("%d",&i)` pour lire au clavier

`stdio` désignant la « sortie standard » qui est l'écran dans notre cas ou l'« entrée standard » qui est le clavier dans notre cas.

Par ailleurs on a aussi la possibilité d'utiliser :

`fprintf(stderr,"Message d'erreur quelconque\n")` pour écrire à l'écran sur la « sortie erreur »

ce qui a l'interêt suivant : durant l'exécution on pourra écrire les résultats normaux sur la sortie standard et les messages d'erreur sur la sortie erreur. A l'aide de Linux il sera possible ensuite de séparer les deux sorties afin que les messages d'erreur ne soient pas disséminés au milieu des résultats.

Relisons le fichier *carres.res* précédemment constitué et écrivons son contenu à l'écran :

```
/* Lecture du fichier carres.res en sachant a priori qu'il contient n valeurs */
#define n 8
#include<stdio.h>
int main(void)
{
int i,ic;
FILE *fich;
fich=fopen("carres.res","r");
for(i=1; i<=n; i++) {fscanf(fich,"%d",&ic); printf("%d\n",ic);}
return (0);
}
```

Cette façon de faire suppose qu'on connaît à l'avance le nombre de valeurs à lire dans le fichier *carres.res*. Si ce n'est pas le cas on utilise la valeur de `fscanf(fich,"%d",&ic)` qui, comme pour `scanf` est le nombre de valeurs qui ont pu être lues correctement. Lorsqu'elle arrive à la fin du fichier, `scanf` ne trouve pas ce qu'elle attend mais une « fin de fichier » et a alors la valeur contenue dans la variable prédéfinie qui se nomme `EOF` (et qui, dans le cas de notre compilateur vaut -1). Il suffit donc de tester la valeur de `fscanf` comme dans l'exemple suivant :

```
/* Lecture du fichier carres sans connaitre a priori le nombre de valeurs qu'il contient */
#include<stdio.h>
int main(void)
{
int ic;
FILE *fich;
fich=fopen("carres.res","r");
while(fscanf(fich,"%d",&ic)!=EOF) printf("%d\n",ic);
return (0);
}
```

4.3 Les différents modes d'ouverture

On a vu jusqu'à présent deux options d'ouverture des fichiers :

en écriture `"w"`
en lecture `"r"`

La liste suivante précise et complète ces options :

`"w"`	écriture seule	le fichier est créé s'il n'existait pas, sinon l'écriture se fait en écrasant ce qui était déjà écrit
`"r"`	lecture seule	le fichier doit déjà exister sinon `fopen` renvoie une erreur
`"a"`	écriture en ajout	le fichier est créé s'il n'existait pas, sinon l'écriture se fait à la suite de ce qui était déjà écrit

4.4 Utilisation de la valeur de `fopen`

Si l'ouverture du fichier se passe bien, on a vu que `fopen` a pour valeur la désignation du fichier dans le programme C. Sinon (cas de l'option `"r"` lorsque le fichier n'existe pas déjà), il a la valeur contenue dans la variable prédéfinie `NULL`.

Exemple

Supposons que le fichier *toto* existe, que *tutu* n'existe pas. L'exécution de :

```
#include<stdio.h>
int main(void)
{
if(fopen("toto","r")==NULL) printf("L'ouverture de toto a échoué\n");
else                        printf("L'ouverture de toto a réussi\n");
if(fopen("tutu","r")==NULL) printf("L'ouverture de tutu a échoué\n");
else                        printf("L'ouverture de tutu a réussi\n");
return (0);
}
```

donne le résultat :
L'ouverture de toto a réussi
L'ouverture de tutu a échoué

Il faut faire ce type de test quand on ne sait pas à priori si le fichier que l'on souhaite ouvrir en lecture existe.

5 Application à la constitution de fichiers de données pour le programme graphique *Gnuplot*

Pour tracer des courbes ou des surfaces on utilisera *Gnuplot*.

5.1 Tracé de courbes planes

Pour tracer des courbes planes, *Gnuplot* peut, entre autres, travailler avec un fichier de points de la forme :

0 0.2
0.04 0.206543
0.08 0.213383
0.12 0.220536
...

contenant les deux coordonnées d'un point par ligne. Si on veut tracer une courbe dont les points sont les résultats d'un calcul effectué par un programme C il suffit d'écrire les coordonnées de ces points selon cette présentation dans un fichier de données pour *Gnuplot*. C'est ce que fait le programme suivant sur l'exemple de la fonction[12] :

$$y = \frac{1}{1+x^2}$$

```
#define N 100
#include<stdio.h>
int main(void)
{
int i; double x,y,dx=0.05;
FILE *fich;
fich=fopen("courbe.don","w");
for(i=-N/2; i<=N/2; i++) {x=i*dx; y=1./(1+x*x); fprintf(fich,"%lg %lg\n",x,y);}
fclose(fich);
return (0);
}
```

Le C a terminé son travail, et, ensuite, pour faire tracer la courbe, le plus simple est d'écrire les quelques commandes *Gnuplot* nécessaires dans un fichier nommé par exemple *courbe.gnu*, soit :

plot 'courbe.don' with lines
pause -1

et on fait exécuter ces commandes en tapant sur la ligne de commande Linux :

gnuplot courbe.gnu

ce qui ouvre une fenêtre *Gnuplot* sur laquelle apparaît la figure suivante :

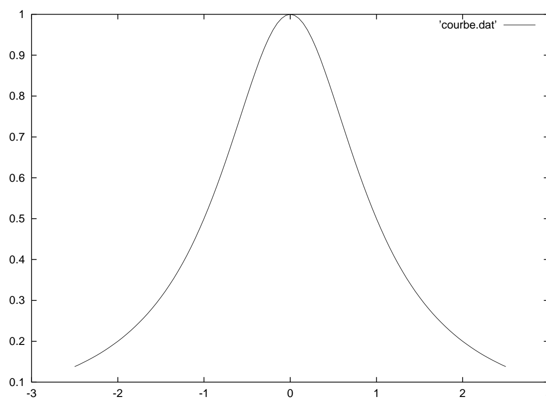

FIG. 1 –

Si on veut sauvegarder cette figure dans un fichier *Postscript* et pouvoir l'imprimer sur papier, on complète le fichier *courbe.gnu* par les commandes au delà des pointillés :

plot 'courbe.dat' with lines
pause -1
...
set term postscript
set output 'courbe.ps'
plot 'courbe.dat' with lines

et la courbe peut être imprimée par la commande Linux *lpr* :

lpr courbe.ps

Remarques

[12] dans cet exemple la fonction est suffisamment simple pour que *Gnuplot* puisse le faire tout seul mais on verra des exemples de fonctions dont le calcul est hors de portée de *Gnuplot*

on verra comment, à l'aide de Linux, tout faire en une seule commande, depuis la compilation du C jusqu'à l'affichage de la courbe sur l'écran et, éventuellement, son impression sur papier.

on pourrait faire exactement la même chose pour une courbe dans l'espace, il suffirait de mettre trois coordonnées par ligne au lieu de deux dans le fichier *courbe.don* et de modifier légèrement *courbe.gnu*[13].

5.2 Tracé de surfaces dans l'espace

De même que pour les courbes, *Gnuplot* peut tracer des surfaces à partir d'un fichier de points. On suppose que la surface est définie par une fonction $z = f(x, y)$ et que les valeurs de x et y forment un réseau rectangulaire régulier dont les points sont indicés par i pour x et j pour y. Il suffit alors d'écrire un fichier dans lequel figurent uniquement les valeurs de z, rangées dans le bon ordre :

$z(x_1, y_1)$
$z(x_1, y_2)$
...
$z(x_1, y_{n-1})$
$z(x_1, y_n)$
ligne blanche
$z(x_2, y_1)$
$z(x_2, y_2)$
...
$z(x_2, y_{n-1})$
$z(x_2, y_n)$
ligne blanche
...
...
$z(x_m, y_1)$
$z(x_m, y_2)$
...
$z(x_m, y_{n-1})$
$z(x_m, y_n)$

en supposant que i varie de 1 à m et j de 1 à n. Soit par exemple :

0.0627905
0.0583811
...
0.0583811
0.0627905
ligne blanche
0.125333
0.116532
...
0.116532
0.125333
ligne blanche
...

Le programme suivant constitue ce fichier pour la fonction[14] :

$$z = \sin x \cos 3y$$

```
#define N 50
#include<stdio.h>
#include<math.h>
int main(void)
{
int i,j; double x,y,dx,dy,z,pi;
FILE *fich;
pi=acos(-1.); dx=pi/N; dy=2*pi/N;
```

[13]des informations plus complètes sur l'utilisation de *Gnuplot* sont données dans un polycopié spécialisé
[14]qui elle aussi, pourrait être traitée directement par *Gnuplot*

```
fich=fopen("surface.dat","w");
for(i=0; i<=N; i++)
   {
   x=i*dx;
   for(j=0; j<=N; j++) {y=j*dy; z=sin(x)*cos(3*y); fprintf(fich,"%lg\n",z);}
   fprintf(fich,"\n");
   }
fclose(fich);
return (0);
}
```

Le fichier *surface.gnu* peut s'écrire :

set hidden3d
set contour
set noparametric
set data style line
splot 'surface.dat'
pause -1
set term postscript
set output 'surface.ps'
splot 'surface.dat'

et donne le résultat :

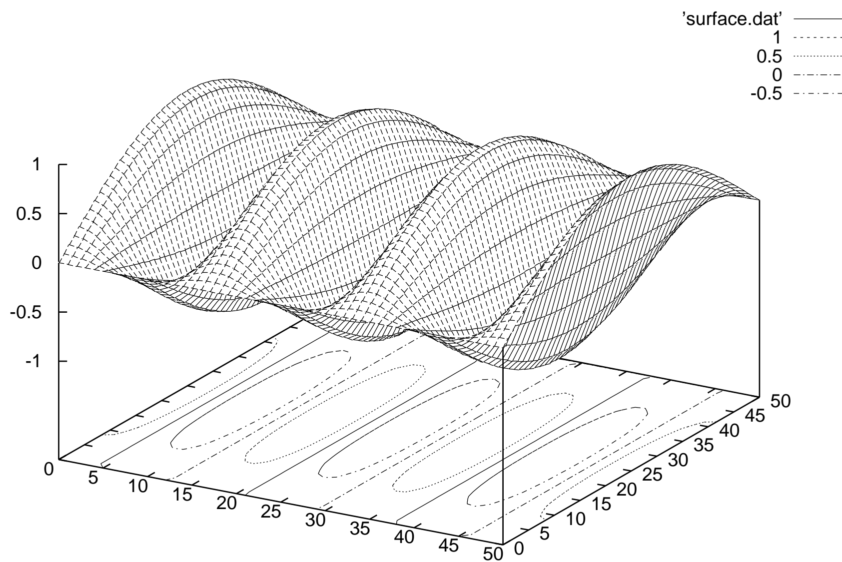

FIG. 2 –

Annexe A : comparaison des formats e f g

Le programme :

```
/* comparaison des formats e f g */
#include<stdio.h>
int main(void)
{
int i; double x=1234567.;
printf("\n  Format e :      Format f :      Format g :\n\n");
for(i=1; i<=11; i++)
   {
     x=x/10.;
     printf("%13.6le   %13.6lf   %13.6lg\n",x,x,x);
   }
return (0);
}
```

donne :

```
  Format e :      Format f :      Format g :

 1.234567e+05   123456.700000         123457
 1.234567e+04    12345.670000        12345.7
 1.234567e+03     1234.567000        1234.57
 1.234567e+02      123.456700        123.457
 1.234567e+01       12.345670        12.3457
 1.234567e+00        1.234567        1.23457
 1.234567e-01        0.123457       0.123457
 1.234567e-02        0.012346      0.0123457
 1.234567e-03        0.001235     0.00123457
 1.234567e-04        0.000123    0.000123457
 1.234567e-05        0.000012    1.23457e-05
```

Annexe B : caractères séparateurs

fin de ligne	*formfeed*	`\n`
tabulation horizontale	*horizontal tab*	`\t`
tabulation verticale	*vertical tab*	`\v`
retour en arrière	*backspace*	`\b`
retour chariot	*carriage return*	`\r`
saut de page	*formfeed*	`\f`
signal sonore	*audible alerte*	`\a`