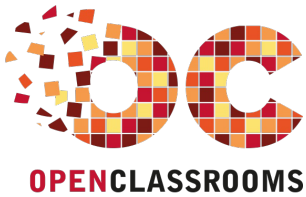
Lecture et écriture de fichiers en mode binaire

Par uknow



www.openclassrooms.com

Dernière mise à jour le 31/08/2010

Sommaire

Sommaire 2
Lecture et écriture de fichiers en mode binaire 3
Le mode formaté et non formaté 3
printf () et le mode formaté 3
La différence entre fwrite et printf 4
Avantages de fwrite par rapport à fprintf 5
Utilisation de fwrite 5
Ecriture d'une variable dans un fichier 6
Ecriture d'un tableau alloué statiquement dans un fichier 7
Ecriture d'un tableau alloué dynamiquement dans un fichier 8
Ecriture d'un pointeur dans un fichier 8
Ecriture d'une structure dans un fichier 9
Les cas dans lesquels fwrite est déconseillée 11
Lecture par fread 11
Pour une variable 12
Pour un tableau statiquement alloué 12
Pour un pointeur 13
Pour un tableau à une dimension, alloué dynamiquement 13
Pour une structure 14
Exemples d'application 14
Portabilité 15
Partager 15

Lecture et écriture de fichiers en mode binaire

Par  uknow

Mise à jour : 31/08/2010
Difficulté : Facile Durée d'étude : 1 heure

Comme ce que vous avez pu voir dans le cours de M@teo21 sur l'écriture et la lecture d'un fichier, si ce n'est pas encore fait, alors je vous le conseille vivement avant de continuer à lire ce présent tutoriel , on peut écrire et lire dans un fichier au format texte (ASCII). Ici je vais vous parler d'une autre méthode de lecture et écriture non formatée qu'on appelle ***écriture binaire***.

En vous souhaitant bonne lecture.
Sommaire du tutoriel :

- Le mode formaté et non formaté
- Avantages de fwrite par rapport à fprintf
- Utilisation de fwrite
- Les cas dans lesquels fwrite est déconseillée
- Lecture par fread
- Exemples d'application
- Portabilité

Le mode formaté et non formaté

printf () et le mode formaté

Je vais parler un peu de printf , et oui ça vous rappelle quelque chose (enfin je l'espère).

Cette merveilleuse fonction qui se tape un grand travail afin de nous afficher les choses qu'on lui donne au format qu'on désire, et ceci en mode dit "formaté", c'est-à-dire, que la donnée que nous lui avons passée sera écrite d'une façon personnalisée et au format ASCII.

Code : C

```
int var = 15;
printf("la valeur de var est %d",var);
```

Ainsi, la valeur de la variable `var` , sera inscrite en caractères ASCII ('1' et '5') sur l'écran.

Code : Console

```
la valeur de var est 15
```

La différence entre fwrite et printf

Maintenant que nous savons un peu ce qu'est l'écriture en mode formaté, voyons ce que c'est qu'un mode dit "non formaté". En l'occurrence le mode d'écriture de fwrite.

Vous avez dû lire dans le cours de M@teo21 que quand on déclare une variable, cela alloue une place dans la mémoire pour contenir ce qu'on va stocker dans cette variable.

Je m'explique

Si je continue sur l'exemple d'avant, une déclaration comme ceci :

Code : C

```
int var;
```

Va allouer un espace qu'on nommera 'var', ainsi si on lui affecte une valeur, cette valeur sera inscrite dans cet espace mémoire, qui, dans le cas d'un int, est généralement sur 4 octets (32 bits).

La notions de bits est importante ici .

Ainsi le résultat d'une affectation comme ceci :

Code : C

```
int var = 15;
```

Donnerait en mémoire :

00000000	00000000	00000000	00001111

Qui est la représentation binaire du nombre 15 sur 32 bits. J'ouvre une parenthèse pour vous informer que l'ordre de ces 4 octets peut différer d'une machine à l'autre selon le codage utilisé (Endianess). Je vais donc vous demander de supposer qu'on est sur une machine utilisant ce codage (ce sera transparent pour la suite, du temps qu'on ne change pas de machine).

C'est quoi le rapport avec fwrite ?

fwrite est une fonction qui se fiche de la valeur enregistrée dans notre variable var, contrairement à printf qui pour afficher '1' et '5' a dû évaluer cette valeur binaire en décimale.
La variable 'var' pour la fonction fwrite sera ni plus ni moins qu'une suite d'octets en mémoire, une utilisation de cette dernière va inscrire dans un fichier les octets représentant notre variable en mémoire.

Si je continue sur l'exemple, la fonction fwrite inscrira ceci dans le fichier :

octet1 - octet2 - octet3 - octet4
00000000 00000000 00000000 00001111

Qui représente notre espace mémoire 'var' contenant la valeur 15.

Avantages de fwrite par rapport à fprintf

L'avantage que fwrite peut avoir par rapport à fprintf, est la simplicité de l'écriture et encore plus, de la lecture . Pour mettre ceci en évidence je vais prendre l'exemple d'une structure.

Si j'ai la structure suivante :

Code : C

```
typedef struct {

     int age;
     char nom[30];
     char prenom[30];
     char adresse[60];
     int nombreFreres;

}SPersonne ;

SPersonne personne;  //Je déclare une variable de type SPersonne
```

Et que je souhaite sauvegarder les données relatives à une personne que j'ai créée. Avec fprintf, je vais être obligé d'écrire champ par champ alors que avec fwrite une seule ligne suffirait pour sauvegarder une personne, et pour la charger aussi . Je vous laisse en juger vous même :

Avec fprintf :

Code : C

```
fprintf( fichier , "%d\n" , personne.age);
fprintf( fichier , "%s\n" , personne.nom);
fprintf( fichier , "%s\n" , personne.prenom);
fprintf( fichier , "%s\n" , personne.adresse);
fprintf( fichier , "%d\n" , personne.nombreFreres);
```

Avec fwrite :

Code : C

```
fwrite( &personne , sizeof(personne) , 1 , fichier);
```

Nous verrons tous ça plus en détail plus loin .

Utilisation de fwrite

Comment s'utilise fwrite ?

Je rappelle que le prototype de cette fonction est :

Code : C

```
size_t fwrite (const void *ptr, size_t size, size_t nmemb, FILE
*stream);
```

- ***ptr*** pointeur sur le premier octet de la liste d'objets à inscrire.
- ***size*** L'espace mémoire pris par un membre de la liste d'objets à inscrire.
- ***nmemb*** Le nombre de membres ayant la taille *size* dans la liste d'objets à inscrire.
- ***stream*** Pointeur sur le flux (pointeur sur FILE dans notre cas).
- **Valeur retournée** La fonction fwrite retourne le nombre d'éléments qu'elle a réussi à inscrire correctement dans le flux pointé par *stream*.

Ecriture d'une variable dans un fichier

Admettons que je veuille sauvegarder ma variable 'var' dans mon fichier, j'utiliserai donc fwrite ainsi :

Code : C

```
int var = 15;

fwrite( &var , sizeof(int) , 1 , fichier);
```

D'où vient le 'fichier' ?

Et bah le 'fichier' c'est le fichier dans lequel je souhaite sauvegarder ma variable, que je dois avoir ouverte préalablement à l'aide de fopen comme ceci :

Code : C

```
FILE * fichier;

fichier = fopen("monfichier.bin" , "wb");
```

Il faut noter les choses suivantes avec la légende "très important" :

- L'extension du fichier n'a pas d'importance, mais ici j'ai choisi le .bin, pour éviter le .txt. Car, contrairement au mode formaté qu'on a vu précédemment, un fichier écrit en mode binaire **ne doit pas** être ouvert ou édité avec un éditeur de texte classique (Bloc-notes par exemple), mais par un éditeur de fichiers binaires.
  De toute façon, le contenu de notre fichier ne sera pas exploitable généralement
- `"wb"` : le 'b' ici indique à fopen qu'on souhaite ouvrir le fichier en mode binaire (donc non formaté), ce même 'b' peut être combiné avec tous les autres modes de la fonction fopen (a,r,w,...).
  Dans le cas de r+,a+ ou w+, le 'b' doit être entre la lettre et le signe '+' comme ceci : "rb+".
- Et le plus important, **toujours** tester le retour de fopen

Maintenant que nous savons ouvrir un fichier en mode binaire, analysons la ligne : `fwrite( &var , sizeof(int) , 1 , fichier);`

1- J'appelle ma fonction fwrite.
2- Je lui donne un pointeur sur l'espace mémoire que je cherche à sauvegarder.
3- Je lui dis que cet espace fait (`sizeof(int)`) 4 octets dans notre cas.
4- Je lui dis qu'il n'y a qu'un seul élément.
5- Et je lui dis que c'est dans 'fichier' que je voudrais sauvegarder tout ça .

Ecriture d'un tableau alloué statiquement dans un fichier

Si j'ai maintenant un tableau de int comme ceci :

Code : C

```
int tab[10];
```

Et que je veuille le sauvegarder dans un fichier alors c'est simple , il suffit de faire ceci :

Code : C

```
fwrite( tab , sizeof(int) , 10 , fichier);
```

Et oui pas besoin de faire une boucle pour inscrire mon tableau case par case .
Car on a demandé à la fonction fwrite d'inscrire l'espace mémoire pointé par 'tab', et dont la portée est 10 x 4 octets. Ce qui correspond aux 10 éléments de mon tableau.

Je vais vous conseiller une autre façon de le faire, que si vous décidez de changer la taille ou le type de votre tableau, cela ne vous obligera pas à changer l'appel à la fonction fwrite. L'écriture est la suivante :

Code : C

```
fwrite( tab , sizeof(tab[0]) , sizeof(tab)/sizeof(tab[0]) ,
fichier);
```

- `sizeof (tab[0])` nous indiquera la taille de chaque élément de notre tableau (4 octets).
- `sizeof (tab)` nous indiquera la taille totale allouée à notre tableau (40 octets), que si on divise par la taille de chaque élément, ceci nous donne le nombre d'éléments que contient notre tableau (10 éléments).

Ceci indépendamment du type de notre tableau et de sa taille

Cette méthode est valable aussi pour des tableaux à plusieurs dimensions et dont la déclaration est faite statiquement (et non par allocation dynamique)

Ainsi une utilisation comme ceci est correcte :

Code : C

```
int tab[10][10];

fwrite( tab , sizeof(tab) , 1 , fichier);
```

Ce cas est seulement pour une déclaration dite "statique" de notre tableau, **pour un cas d'allocation dynamique, ceci n'est plus valable.**

Ecriture d'un tableau alloué dynamiquement dans un fichier

Si je dispose d'un tableau que j'ai alloué dynamiquement par malloc comme ceci :

Code : C

```
int * ptab;

ptab = malloc(10 * sizeof(int));
//Ne pas oublier de tester le retour de malloc !
```

la sauvegarde dans notre fichier s'effectuera de la même façon qu'un tableau statique :

Code : C

```
fwrite( ptab , sizeof(int) , 10 , fichier);
```

Ou indépendemment du type ainsi :

Code : C

```
fwrite( ptab , sizeof (* ptab) , 10 , fichier);
```

Ceci n'est pas vrai dans le cas de tableaux à plusieurs dimensions (alloués dynamiquement), il faudrait écrire chaque dimensions comme présenté ci-dessus.

Ecriture d'un pointeur dans un fichier

Ce cas est très identique à celui d'un tableau alloué dynamiquement.

Si on a un pointeur 'ptr' ayant une taille allouée de 'size' octets, alors fwrite s'utilise ainsi :

Code : C

```
fwrite ( ptr , size , 1 , fichier);
```

ainsi, tout le bloc mémoire alloué pour notre pointeur sera considéré comme un seul élément uni (d'où le 1 au 3ième argument).

On peu également considérer que la mémoire comporte 'size' éléments de taille 1 octet, auquel cas l'utilisation de fwrite devient :

Code : C

```
fwrite ( ptr , 1 , size , fichier);
```

Il s'agit de deux notions différentes, et qui aboutiront à deux façons parfois différentes de construire notre fichier. Alors dans la lecture avec fread, on fera attention à respecter ce détail.

Ecriture d'une structure dans un fichier

L'écriture d'une structure est très similaire aux cas présentés ci-dessus, car pour fwrite, encore une fois, la structure ne sera qu'une suite d'octets. Cependant, il y a quelques petites notions à comprendre  . D'ailleurs c'est pourquoi je fais ce tutoriel.

Si nous avons une structure comme ceci :

Code : C

```
typedef struct {
    int age;
    char nom[30];
    char prenom[30];
}Personne;
```

Alors il n'y aucun problème à l'utilisation de l'opérateur **sizeof** pour savoir la taille de notre structure.

Code : C

```
Personne personne1 = {15,{"NOM"},{"Prenom"}};

fwrite( &personne1 , sizeof(personne1) , 1 , fichier );
```

Maintenant si on a une structure comme ceci :

Code : C

```
typedef struct {
    int age;
    char * nom;
    char * prenom;
}Personne;

Personne personne1 = {15,"TOTO","TATA"};
```

Alors si on essaie de récupérer la taille de cette dernière par un sizeof, ceci nous donnera la taille de age + la taille de 'nom' + la taille de 'prenom'.

Où est le problème ?

Le problème est que 'nom' et 'prenom' sont deux variables de type pointeur. Et leurs tailles sont les tailles d'un pointeur (généralement 4 octets) et non celles des chaines de caractères sur lesquelles ils pointent.

ainsi le résultat de `sizeof(prsonne1)` donnerait 12 octets (4 + 4 + 4) quelque soient les chaines sur lesquelles ils pointent.

```
fwrite( &personne1 , sizeof(personne1) , 1 , fichier );
```

Si on essaie malgré cela, d'utiliser fwrite comme indiqué ci-dessus, cela va sauvegarder les adresses des chaines pointées par 'nom' et 'prenom'. Qui seront, après la fermeture du programme, non signifiantes. Et leur utilisation aboutirait à un SEGFAULT ou ACCESS VIOLATION à coup sûr (sauf cas de chance).

Comment faire alors dans de tels cas ?

2 solutions sont possibles :

Solution 1 :

Inscrire séparément l'age, le nom et le prénom. Dans ce cas je vous conseille vivement d'utiliser fprintf comme vous l'avez appris

Solution 2 :

Procéder à une sérialisation de nos données (age, nom et prénom).

C'est quoi une sérialisations ?

Une sérialisation de données est un terme informatique qui consiste à mettre toutes nos données en série (les unes à la suite des autres) dans un seul buffer pour être envoyées par réseau ou être inscrites dans un flux. Ceci peut être vu comme une concaténation .

Donc une sérialisation de notre structure donnerait :

4 octets	5 octets	5 octets

15	"TOTO\0"	"TATA\0"

Ce buffer sera donc à enregistrer dans le fichier par fwrite.

Je ne rentre pas dans les détails car ceci impliquerait l'écriture d'un tutoriel. Si vous êtes intéressés, vous pouvez faire des recherches sur les techniques de sérialisation de données .

En conclusion :

L'enregistrement en mode binaire, n'est pas adéquat avec de telles déclarations. Pour y parvenir il faut, quand vous en avez la possibilité, déclarer statiquement les champs d'une structure pour pouvoir la sauvegarder en mode binaire (avec fwrite) sans problèmes et sans manœuvres particulières (sérialisation ou autres).

Mon but n'étant pas de vous inciter à déclarer toujours statiquement vos données (tableaux en particulier), car très souvent on déclare plus qu'on en a besoin. Ce qui n'est pas très optimisé.

Donc à vous de trouver le compromis idéal pour votre application, entre la simplicité de sauvegarde, et l'optimisation de mémoire.

Les cas dans lesquels fwrite est déconseillée

Dans l'un des cas suivants, il est déconseillé d'utiliser fwrite (pour un débutant bien évidemment) :

- Celui présenté juste au dessus. A savoir une structure comportant un ou plusieurs pointeurs dans ses champs :

Code : C

```
typedef struct {
    int age;
    char * nom;
    char * prenom;
}Personne;
```

- Si l'on souhaite éditer nos données avec un éditeur de texte du genre bloc-notes par exemple, car ce ne sera pas possible pour un fichier créé en mode binaire. Il faudra donc utiliser un éditeur de fichiers binaires.

- Si à la déclaration de vos structures et vos variables, vous n'avez pas prévu une manoeuvre de sauvegarde dans un fichier en mode binaire, alors je vous déconseille de vouloir le faire à tout prix, car généralement cela implique un bricolage (des sérialisations, des changements de types...) bref beaucoup de bidouilles qui rendent votre code incompréhensible, et pas facil à déboguer .

Lecture par fread

Il est important de noter que la lecture par fread doit correspondre parfaitement à la manière dont on a écrit avec fwrite. Et que pour pouvoir lire un fichier créé sur une autre machine, il faut que ces deux machines aient utilisé le même encodage notamment l'Endianess (little, big, bi ou middle).

Par conséquent, fread ne pourra pas être utilisée si le fichier à été créé avec les fonctions fputs, fprintf ou les autres fonctions écrivant en mode formaté.
Ou si le fichier est édité (rempli) avec un éditeur de texte classique (type bloc-notes), sauf éditeurs binaires, auquel cas il faut maîtriser ce que l'on écrit.

L'utilisation de fread est aussi simple que fwrite, et s'utilise de la même manière, je rappelle le prototype de la fonction fread :

Code : C

```
size_t fread (const void *ptr, size_t size, size_t nmemb, FILE
*stream);
```

- ***ptr*** pointeur sur le premier octet de la liste d'objets à charger.
- ***size*** L'espace mémoire pris par un membre de la liste d'objets à charger.
- ***nmemb*** Le nombre de membres ayant la taille *size* dans la liste d'objets à charger.
- ***stream*** Pointeur sur le flux (pointeur sur FILE dans notre cas).
- **Valeur retournée** La fonction fread retourne le nombre d'éléments qu'elle a réussis à lire correctement dans le flux pointé par *stream*. Il faut noter aussi que la fonction fread traite la fin du fichier comme une erreur de lecture.

L'utilisation étant similaire à fwrite ainsi que les conditions d'utilisation présentées ci-dessus, alors je ne tarderai pas dans les explications :

Pour une variable

Si l'on a une variable 'var' qu'on a sauvegardée par fwrite dans le fichier 'fichier' alors on peut la charger ainsi :

Code : C

```
int var;

fread( &var , sizeof(var) , 1 , fichier );
```

Pour un tableau statiquement alloué

L'utilisation est identique à celle décrite pour fwrite , et est valable pour des tableaux multidimensionnels statiquement alloués.

Code : C

```
int tab[10];

fread( tab , sizeof(tab) , 1 , fichier );
```

Si on a sauvegardé avec `fwrite( tab , sizeof(tab) , 1 , fichier );`

Et

Code : C

```
fread( tab , sizeof(tab[0]) , sizeof(tab)/sizeof(tab[0]) , fichier
);
```

Si on a sauvegardé avec `fwrite( tab , sizeof(tab[0]) , sizeof(tab)/sizeof(tab[0]) , fichier );`

La différence entre ces deux notions se situe dans la disposition différente des éléments dans le fichier, selon si on a utilisé une méthode ou une autre.
Donc il faut respecter la méthode avec laquelle on a effectué l'écriture. Et ne pas mélanger les deux.

Et pour un tableau à deux dimensions :

Code : C

```
int tab[10][10];

fread( tab , sizeof(tab) , 1 , fichier );
```

pratique n'est-ce pas  ?

Pour un pointeur

La condition à utiliser fread est **qu'on ait alloué préalablement de l'espace** pour notre pointeur.

Code : C

```
ptr = malloc(sizeof * ptr);
//Ne pas oublier de tester le retour de malloc :)

fread( ptr , sizeof * ptr , 1 , fichier );
```

Pour un tableau à une dimension, alloué dynamiquement

C'est le même cas qu'un pointeur. A savoir **qu'il faut allouer au préalable assez d'espace pour lire les éléments du fichier.**

Code : C

```
int * ptableau;
ptableau = malloc(nombreElements * sizeof *ptableau);
//Ne pas oublier de tester le retour de malloc

fread( ptableau , sizeof * ptableau , nombreElements , fichier );
```

Pour le cas d'un tableau à plusieurs dimensions il faut procéder par boucle en lisant dimension par dimension pour arriver au cas décrit ci-dessus!

Pour une structure

Identiquement à fwrite si l'on a une structure déclarée ainsi :

Code : C

```
typedef struct {
    int age;
    char nom[30];
    char prenom[30];
}Personne;

Personne personne1;
```

Alors la lecture s'effectue comme ceci :

Code : C

```
fread( &personne1 , sizeof(personne1) , 1 , fichier );
```

Dans le cas d'une écriture après une sérialisation, il faut récupérer les données dans un buffer et procéder par utilisation de memcpy à une "désérialisation".
Je n'explique pas plus car ceci n'est pas le sujet  .

Exemples d'application

En exemple d'application on pourrait citer la sauvegarde des données d'un joueur, il est donc conseillé de prévoir ceci en déclarant la structure contenant les informations à sauvegarder  .

Code : C

```
typedef struct {
     char nom[30];
     int niveau;
     int force;
     int vies;
     Map dernierePartie;
     //......
}Joueur;
```

Et veiller à, quand vous en avez la possibilité, déclarer statiquement les champs de cette structure. Ainsi la sauvegarde et le chargement ne vous couteront qu'une ligne de code chacune  . Même pour le cas de plusieurs joueurs.
Ceci représente l'inconvénient majeur, car très souvent, on ne sait pas à priori, qu'elle taille on doit avoir pour stocker une donnée. Dans ce cas l'allocation dynamique s'impose.

Un deuxième exemple est la configuration d'une application que vous avez développée.
Des données telles que, des chemins d'accès, temps, date....

Ou faire une base de donnée sans avoir à aller lire ligne par ligne et identifier des séparateurs que vous aurez mis entre deux données etc...
Imaginons que j'aie un tableau contenant mes données :

Code : C

```
Data donnees[50];
```

Au lieu d'aller traiter chaque champ de chaque donnée, j'aurais préféré un seul coup de fwrite pour tout sauvegarder et un coup de fread pour tout charger .

Portabilité

Cette partie du tutoriel n'est pas là pour vous donner des solutions portables, mais uniquement pour vous sensibiliser aux différents problèmes de portabilité que vous pouvez rencontrer.

Voici les problèmes auxquels vous serez **peut être** confrontés si vous désirez lire un fichier binaire créé sur **une autre machine**.

- Le premier étant la taille des types. Les 4 octets de notre `int` (présenté en haut) peuvent varier d'une implémentation à une autre. Donc pensez à utiliser les types standards pour un int (uint_8, uint_16, uint_32, uint_64...). Pour les autres types vous pouvez consulter la description de la norme.
- Le type de codage utilisé, l'"endianess", qui définit une disposition différente des octets formant un objet en mémoire, et de leurs poids significatifs.
- Dans le cas d'une structure, la disposition des champs en mémoire peut varier d'une machine à l'autre. Notamment dans le cas d'un int qui nécessite une taille multiple de 2 ou de 4, l'implémentation peut donc faire appel à des bit dits de "bourrage", pour des raisons de performance (notamment la manipulation de données en mémoire par le CPU).
- En ce qui concerne l'aspect contigüe des champs formant la structure, certains disent que cette mémoire n'est pas garantie d'être contigüe (que les champs de la structure ne sont pas disposés les uns à la suite des autres en mémoire). La norme dit :

  Citation : 6.2.5 Types

  > A structure type describes a sequentially allocated nonempty set of member objects
  > (and, in certain circumstances, an incomplete array), each of which has an optionally specified name and possibly distinct type.

  "Un type structure décrit une séquence non vide allouée..."Donc ceci est en quelque sorte vrai, car la norme ne dit pas explicitement que la mémoire sera contigüe (comme ce qu'elle fait avec les tableaux). Mais elle utilise le mot séquence qui réfère à une suite de données (les unes à la suite des autres) mais sous-entend la possibilité de trouver des bits de bourrage (ce qu'on appelle "*padding*") entre deux données successives. Ceci étant pour des raisons d'optimisation (de performance CPU), car ceci lui permet de gérer plus facilement des données selon si elles sont disposées d'une façon ou d'une autre.

Je vous rappelle que cette partie ne doit être prise en compte que dans le cas de changement de machine entre l'écriture d'un fichier binaire et sa lecture (ou la compilation sur deux implémentations différentes). Donc si vous travaillez sur une seule machine, tous ces problèmes ne se poseront pas. Et c'est ce qu'on appelle **la non-portabilité** .

La portabilité des fichiers binaires est un domaine très vaste, j'ai donc expliqué ici très brièvement une partie des problèmes que l'on peut rencontrer . Si vous êtes intéressés je vous invite à faire plus de recherches en vous aidant du draft de la norme ISO/IEC 9899 (la version 1124 étant la plus récente facilement trouvable sur le net).
Vous savez à présent quels sont les avantages et inconvénients d'une manipulation binaire de fichiers. Donc faites-en bon usage .