Les erreurs courantes en C

par Nicolas Joseph

Date de publication : 12 Septembre 2005

Dernière mise à jour :

Ce document regroupe une liste des erreurs les plus couramment rencontrées par un programmeur en C, ainsi que des bonnes habitudes à prendre pour éviter d'obtenir des comportements indéfinis

Confusion entre == et =

Pour vérifier une égalité, il faut utiliser l'opérateur == (constitué de deux signes égal) hors il est très facile de n'en taper qu'un seul. Ceci a pour conséquence de réaliser une affectation :

```
if (size = 0)
```

Dans ce cas, l'expression retourne 0, donc le bloc **if** ne sera jamais exécuté. Voici deux solutions possibles :

Confusion entre le ET logique et le ET binaire

Il est facile de confondre le ET logique (&&) qui retourne 0 ou 1 (en s'arrêtant au premier argument s'il est faux) et le ET binaire (&) qui, quant à lui, évalue ses deux opérandes. Cette confusion est plus difficile à détecter que la précédente puisque même un compilateur bien réglé ne vous avertira pas de l'erreur.

Confusion entre le OU logique et le OU binaire

Le problème est similaire au précédent sauf que cette fois ci, il porte sur le OU logique (||) et sa version binaire (|). Le second opérande du OU logique n'est pas évalué si le premier est vrai.

Problème de boucle

Une erreur aussi courante et tout aussi difficile à détecter que les précédentes, il s'agit du point virgule en trop au niveau de l'instruction de boucle :

```
int i;
for (i = 0; i < 10; i++);
   printf ("%d\n", i);
```

Dans ce cas, la boucle **for** sera exécutée puis *i* sera affiché avec la valeur de sortie de la boucle (10 dans notre cas). L'erreur est encore plus gênante avec une boucle **while** puisque le programme peut entrer dans une boucle sans fin.

Fonctions retournant un caractère (*getc*)

Les fonctions **fgetc**, **getc** et **getchar** permettent de récupérer un caractère sur un flux d'entrée (*stdin* pour **getchar**). Ces fonctions retournent un **int** et c'est à ce niveau que réside le problème en particulier lorsque la variable utilisée pour stocker le retour de ces fonctions est un **char** :

```
char c;
while ( (c = getchar ()) != EOF)
  ...
```

Pour expliquer d'où vient le problème, voici les cast implicites réalisés par le compilateur :

```
while ( (int)(c = (char)getchar ()) != EOF)
  ...
```

Or le caractère EOF (qui marque la fin d'un fichier : **E**nd **O**f **F**ile) est un caractère invalide généralement égale à -1 mais il peut parfaitement être égal à 128, dans ce cas on dépasse la capacité de stockage d'un **char** et l'on se retrouve avec un résultat égale à **-**128 : la condition du **while** sera toujours fausse, le programme boucle indéfiniment!

Récupérer le texte entré au clavier

La première fonction généralement présentée pour récupérer une saisie de l'utilisateur est **scanf** or il s'agit de l'une des fonctions les plus difficiles à maîtriser en C. Voici le genre de code que l'on peut trouver dans certains livres d'initiation :

```
int nombre;
scanf ("%d", &nombre);
```

Premièrement le risque d'oublier le **&** devant les paramètres de **scanf** est assez important. Deuxièment que se passe-t-il si l'utilisateur entre autre chose qu'un entier (du texte par exemple)? On obtient un comportement indéfini! Et dans le cas de la saisie de texte, voici un autre code d'exemple :

```
char texte[20];
scanf ("%s", texte);
```

Tout aussi dangereux! Qu'est ce qui empêche l'utilisateur de saisir plus de 19 caractères? Dans ce cas rienIl existe un moyen de limiter la taille de la chaîne récupérée par **scanf** en utilisant les paramètres de formatage.! La fonction **gets** peut sembler pratique pour récupérer du texte mais l'on retrouve le même problème du contrôle de la taille du texte saisi.

Il reste plus qu'une fonctionRien ne vous empêche de coder votre propre fonction en utilisant **fgetc**, voici une exemple que j'utilise dans mes projets : getNextLine. : **fgets**, c'est la seule fonction qui doit être utilisée pour récupérer une chaîne de caractère. Pour récupérer des nombres, il suffit de convertir le résultat de **fgets** grâce à **strtol** ou **strtod** (**atoi** et **atof** étant obsolètes).

Tester la fin d'un fichier

On trouve souvent ce genre de test :

```
FILE *stream;
...
while (!feof (stream))
...
```

La fonction **feof** n'est pas faite pour cela. Il faut tester le retour de la fonction de saisie (**fgets** ou **fgetc**) qui retourne NULL ou EOF en cas d'erreur ou de fin de fichier et ensuite seulement on peut utiliser **feof** pour connaître la cause de l'arrêt de la lecture du flux.

Vider le buffer d'un flux entrant

La fonction **fflush** permet de vider le tampon des flux sortants et uniquement ceci! Son utilisation dans le cas d'un flux entrant (*stdin* par exemple) entraîne un comportement indefini, on utilisera donc :

```
int ch;
while( (ch = fgetc(fp)) != EOF && ch != '\n' )
  ;
```

Les chaînes de caractères

En C, les chaînes de caractères n'existent pas, il s'agit de tableaux de caractères terminés par un caractère nul (\0). Lors de la création d'une chaîne, il ne faut pas oublier de prévoir une case de plus pour y stocker ce marqueur de fin de chaîne :

```
char src[] = "Developpez";
char *dst = NULL;

dst = malloc (sizeof (*dst) * 10);
strcpy (dst, src);
```

L'utilisation de *dst* comme une chaîne de caractère provoquera un comportement indéfini. Pour les mêmes raisons, il n'est pas possible de comparer deux chaînes grâce à l'opérateur d'égalité (==) :

```
char str1[] = "abc";
char str2[] = "abc";

if (str1 == str2)
  printf ("str1 et identique a str2");
else
  printf ("str1 et different de str2");
```

Ce code affichera toujours que les deux chaînes sont différentes puisque l'on compare les adresses des variables *str1* et *str2*. Pour que le code fasse ce que l'on souhaite, il faut utiliser la fonction **strcmp**.

Pour en finir avec les chaînes de caractères, l'expression :

```
char *texte = "Developpez";
```

N'alloue pas de mémoire pour stocker la chaîne, on se contente de mémoriser l'adresse de la chaîne. Hors rien de garantie que celle-ci est stockée dans une zone de mémoire accéssible en écriture, par précaution, il vaut mieu écrire :

```
const char *texte = "Developpez";
```

Allouer correctement de la mémoire

Pour allouer de la mémoire, il faut utiliser la fonction **malloc** (ou **calloc** au choix) qui demande la taille de la zone à allouer. Il est d'usage courant d'utiliser le type pour déterminer cette taille :

```
int *tab = malloc (sizeof (int) * BUFSIZ);
```

> *Avant de continuer je tiens à signaler qu'il est déconseillé de caster le retour de **malloc**, puisque premièrement c'est inutilLe cast d'un pointeur **void** est inutile en C contrairement au C++ : Les incompatibilités entre le C et le C++ et secondement cela peut cacher l'absence d'inclusion de stdlib.h.*

Il peut arriver que le type de *tab* soit modifié, dans ce cas il faut reprendre tous les appels à **malloc** pour les modifier et en cas d'oubli la sanction peut être lourde de conséquence. C'est pourquoi je vous conseille d'utiliser le nom de la variable pour retrouver la taille de l'élément pointé :

```
int *tab = malloc (sizeof (*tab) * BUFSIZ);
```

Dans ce cas plus de problème! Et si on modifie le nom de la variable? Si vous oubliez un ancien nom dans le fichier source, c'est le compilateur qui se chargera de vous le rappeler.

> *La fonction **free** gère parfaitement le cas où l'adresse qui lui ai passée en argument est NULL, il est donc inutile d'écrire ceci :*

```
if (p != NULL)
  free (p);
```

Tester le retour des fonctions

Certaines fonctions peuvent échouer (je pense en particulier à **malloc** et **fopen**) il est donc bon de tester leurs valeurs de retour pour savoir si l'on peut continuer. Pour illustrer ceci, voici comment utiliser correctement la fonction **realloc** pour ne pas avoir de surpise :

```
void *tmp = NULL;
char *p = NULL;
...
tmp = realloc (p, sizeof (*p) * BUFSIZ);
if (tmp != NULL)
{
   p = tmp;
   /* Le programme continue normalement */
}
else
{
   free (p);
   /* Il faut informer l'utilisateur du manque de
   memoire et quitter proprement l'application */
}
```

Oubli d'un break dans un switch

L'instruction **switch** n'est qu'un **goto** déguisé qui saute vers l'instrcution **case** correspondante et c'est tout! Si vous écrivez ceci :

```
int val = 1;
switch (val)
{
   case 1:
      printf ("val=1\n");
   case 2:
      printf ("val=2\n");
}
```

Le programme affichera les deux textes, pour n'afficher qu'un texte par valeur, il faut ajouter l'instruction **break** :

```
int val = 1;
switch (val)
{
   case 1:
      printf ("val=1\n");
   break;
   case 2:
      printf ("val=2\n");
   break;
}
```

Le second **break** n'est pas indispensable mais cela évitera de l'oublier en cas d'ajout de nouvelles valeurs.

Division entière

Pour réaliser une division entière ou réelle, le C utilise le même opérateur (/). Par conséquent pour savoir quel résultat retourner, le compilateur se base sur le type des opérandes. Voici différents exemples avec le résultat obtenu :

```
double a = 1/2 /* = 0 */
```

```
double b = 1.0/2 /* = 0.5 */
```

```
int a = 1, b = 2;
double c = ((double) a)/b; /* = 0.5 */
```

Remerciements

Merci à Gnux pour la relecture attentive de cet article.