Les nombres aléatoires en C

par Nicolas Joseph (home) (Blog)

Date de publication : 10 Octobre 2005

Au travers de ce tutoriel, je vais vous exposer différentes méthodes pour générer une suite de nombres pseudo-aléatoires. La théorie peut s'appliquer à tous les langages de programmation. Par contre les exemples seront donnés en C.

I - Introduction
II - Les fonctions du C
    II-A - rand
    II-B - srand
III - Une méthode (trop) simple
IV - Mettons-y notre grain de sable
V - Fixons des limites
VI - Jouons à la loterie
VII - Comment calculer le hasard
VIII - Conclusion
IX - Remerciements

I - Introduction

Pour commencer, je tiens à rectifier le titre de ce tutoriel. En effet avec un ordinateur il est impossible de générer une suite de nombres réellement aléatoires, nous devrons nous contenter de nombres pseudo-aléatoires.

Je vais donc vous proposer plusieurs méthodes, de la plus simple à la plus compliquée, pour obtenir une série de nombres difficilement déterminable à l'avance (dite plus communément aléatoire).

II - Les fonctions du C

Avant de nous lancer dans la pratique, voici une brève description des fonctions permettant d'obtenir un nombre pseudo-aléatoire en C. Ces fonctions sont déclarées dans *stdlib.h*.

II-A - rand

Prototype :
```
int rand(void);
```

C'est cette fonction qui retourne un nombre aléatoire à chaque appel. Ce nombre est compris entre 0 et RAND_MAX.

II-B - srand

Prototype :
```
void srand(unsigned int seed);
```

La fonction **srand** permet d'initialiser le générateur de nombres pseudo-aléatoires avec une graine différente (1 par défaut). Elle ne doit être appelée qu'une seule fois avant tout appel à **rand**.

III - Une méthode (trop) simple

Voici une première méthode :

```
int number = rand();
```

Je vous avais prévenu!

Voici un exemple tout simple pour tester nos différentes méthodes :

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int my_rand (void);

int main (void)
{
   int i;

   for (i = 0; i<1000; i++)
   {
     printf ("%d\n", my_rand());
   }
   return (EXIT_SUCCESS);
}

int my_rand (void)
{
   return (rand ());
}
```

*A partir de maintenant, seule **my_rand** sera donnée puisque le reste du programme sera le même.*

Relancez le programme plusieurs fois et observez la suite de valeurs : elle est identique à chaque appel ! Ceci est dû à la graine qui est toujours la même : même graine, même suite de nombres !

On réservera donc cette méthode lorsque l'on a besoin d'un tableau d'entiers, pour éviter d'avoir à le remplir à la main.

IV - Mettons-y notre grain de sable

Vous l'aurez sans doute deviné, pour éviter de retrouver la même suite de nombres à chaque exécution du programme, il faut modifier la graine, et donc appeler **srand** à chaque démarrage du programme avec une graine différente. Il y a une valeur qui est différente à chaque appel du programme : l'heure. En initialisant le générateur avec l'heure actuelle, on devrait obtenir une suite de nombres différente à chaque fois :

```
int my_rand (void)
{
   static int first = 0;

   if (first == 0)
   {
      srand (time (NULL));
      first = 1;
   }
   return (rand ());
}
```

La liste change à chaque appel à condition que intervalle de temps entre deux appels ne soit pas trop court (plus d'une seconde), sinon la suite de nombre sera la même puisque la valeur retournée par **time** sera la même.

V - Fixons des limites

Jusqu'à présent, les valeurs obtenues sont comprises entre 0 et RAND_MAX. Il serait intéressant de limiter l'intervalle de valeurs de 0 à N-1. Pour commencer, une méthode simple consiste à utiliser l'opérateur modulo (extrait de la FAQ **FAQ C**) :

```
#include <stdlib.h>

int randomValue;
randomValue = rand() % N;
```

Cette méthode ne fournit pas une distribution homogène des données (sauf si N est un multiple de RAND_MAX). En effet prenons l'exemple où N est égal à 10 et RAND_MAX à 25 :

N	randomValue
[0;10[	[0;10[
[10;20[	[0;10[
[20;25[	[0;5[

Nous obtenons plus de nombres compris entre 0 et 5, pour pallier ce problème, il faut réaliser une "mise à l'échelle" (extrait de la FAQ **FAQ C**) :

```
#include <stdlib.h>

int randomValue = (int)(rand() / (double)RAND_MAX * (N - 1));
```

VI - Jouons à la loterie

Pour ajouter une dose de hasard notre générateur va, lors du premier appel, créer un tableau de nombres aléatoires, puis à chaque nouvel appel un nombre sera pris au hasard dans ce tableau, sauvegardé pour être retourné par la fonction et pour finir remplacez par un nouveau nombre aléatoire : il s'agit de l'algorithme de C. Bays et S.D.Durham.

```
#define N 100

int my_rand (void)
{
   static int tab[N];
   static int first = 0;
   int index;
   int rn;

   if (first == 0)
   {
      int i;

      srand (time (NULL));
      for (i = 0; i < N; i++)
         tab[i] = rand();
      first = 1;
   }
   index = (int)(rand() / RAND_MAX * (N - 1));
   rn = tab[index];
   tab[index] = rand();
   return (rn);
}
```

Maintenant que nous savons utiliser correctement un générateur de nombres aléatoires, nous allons créer le nôtre.

VII - Comment calculer le hasard

Le titre de ce chapitre reflète bien le problème posé par la génération de nombres aléatoires : comment faire du hasard avec une machine aussi précise qu'un ordinateur ? Au niveau matériel ne cherchez pas : tout est basé sur l'horloge interne, donc réglé comme du papier à musique! Nous allons donc être obligés de créer notre générateur. Comment ? En réalisant différentes opérations sur un nombre de départ (appelé *graine* ou *seed* en anglais) en suivant le principe des suites (rappelez vous vos cours de mathématiques). D'autres se sont posé la question avant nous et en 1948, un certain Monsieur Lehmer a inventé une formule générale de générateur :

$$X_{n+1} = (a * x_n + b) \% c$$

L'opérateur % (modulo) renvoie le reste de la division entière de ses deux opérandes.

Voilà, nous allons pouvoir recréer les fonctions **srand** et **rand** du C !

rand.h
```
#ifndef H_RAND
#define H_RAND

#include <limits.h>

#define RAND_MAX INT_MAX

void rnd_srand (unsigned int);
int rnd_rand (void);

#endif /* not H_RAND */
```

rand.c
```
#include "rand.h"

static int g_seed = 1;

void rnd_srand (unsigned int seed)
{
   g_seed = seed;
   return;
}

int rnd_rand (void)
{
   g_seed = ( 32 * g_seed + 7 ) % 1024;
   return (g_seed);
}
```

Testons :

```
455
231
231
231
231
...
```

C'est plutôt prévisible pour une suite de nombres aléatoires ! Un sujet aussi complexe que le hasard ne peut être résumé par une formule aussi simple. Il existe des contraintes dans le choix des différents paramètres pour éviter ce genre de problème (ce qui, ici, se détecte facilement mais est parfois plus difficilement décelable car visible uniquement pour des valeurs précises de X) :

- **b** et **c** ne doivent pas être multiple l'un de l'autre
- **a**-1 doit être un multiple de **n**, avec **n** tous les nombres premiers diviseurs de **c**
- Si **c** est multiple de 4, **a**-1 doit être un multiple de 4

Même si ces conditions sont réunies, il peut subsister des erreurs, ou plutôt des imperfections au niveau du caractère aléatoire des nombres. Par exemple, si **c** est une puissance de 2, le bit de poids faible des nombres oscillera successivement entre 0 et 1. De même pour le générateur UNIX :

$$X_{n+1} = (1103515245 * x_n + 12345) \% 2147483647$$

Même si ce générateur fonctionne correctement, il faut tout de même faire attention : les octets de poids faibles ne sont pas réellement aléatoires.

VIII - Conclusion

Ce tutoriel vous a présenté les bases de la génération de nombres pseudo-aléatoires. Mais, bien sûr, il existe un nombre bien plus important de générateurs. Vous pourrez en apprendre plus sur le **forum algorithmes** de **developpez.com**.

IX - Remerciements

Merci à 2Eurocents pour la relecture attentive de cet article.