Nombres aléatoires en C et en C++

Jean-Marc Bourguet
jm@bourguet.org

25 février 2009

1 Introduction

La bibliothèque standard du C contient une interface simple vers un générateur de nombres aléatoires (le C++ n'offre rien de plus sauf via le *C++ Library Extensions Technical Report 1* ce dont il ne sera pas question ici). Lors de l'utilisation de celle-ci, il est courant de devoir réduire l'intervalle dans lequel sont générés les nombres. Ce n'est pas très compliqué, mais il faut quand même prendre quelques précautions pour éviter de dégrader inutilement la qualité de la suite générée ou d'entrer dans un comportement indéfini et donc pouvant être surprenant sur certaines plate-formes. Or la majorité des méthode généralement préconisées pour ce faire a l'un ou d'autre de ces défauts. Une meilleure méthode est expliquée ici.

Il est naturellement permis de se demander ce que ça veut dire pour une suite de valeurs déterminée algorithmiquement d'être plus aléatoire que d'une autre... Le débat peut facilement devenir philosophique. Un critère possible est d'avoir des résultats proches de ceux d'une suite aléatoire à une batterie de tests statistiques mais il faut être conscient que pour être significatifs les tests à utiliser doivent être choisis en fonction l'application.

2 Interface

Pour la gestion des nombres aléatoires, l'inclusion de l'en-tête `stdlib.h` définit une macro et deux fonctions.

La fonction principale est `rand()` qui renvoie un **`int`** uniformément réparti entre 0 et `RAND_MAX` (bornes comprises).

`RAND_MAX` est une macro donnant la valeur la plus grande retournée par `rand()`.

`srand(`**`unsigned`** `seed)` est une fonction initialisant le générateur de nombre aléatoire. La séquence des valeurs retournées par `rand()` est déterminée par la valeur utilisée pour `seed`.

3 Initialisation

On n'appelle généralement `srand()` qu'une fois par programme avec comme argument un nombre qui doit être différent à chaque exécution si on désire que la suite de nombres aléatoires générée ne soit pas la même pour toutes les exécutions. La manière la plus portable est d'utiliser le résultat de `time()` (défini dans `time.h`) de la manière suivante :

```
srand((unsigned)time(NULL));
```

Cette méthode a l'inconvénient de générer la même séquence pour des lancements du programme faits dans la même seconde, ce qui est parfois gênant lors de petits exemples et peut aussi l'être pour des programmes plus importants dont plusieurs exécutions peuvent être lancées simultanément sur plusieurs machines. Si on accepte d'être dépendant de la plate-forme, on peut utiliser une source de temps plus précise et la combiner avec des informations, telle qu'un identificateur de processus ou de machine ; certaines plates-formes ont aussi des sources de nombres réellement aléatoires qui peuvent servir pour cela [1].

Si généralement on désire que chaque exécution profite d'une suite de nombres aléatoires différente, il est souvent utile de pouvoir aussi générer à nouveau une suite de nombres précédemment utilisée (pour s'assurer qu'on teste bien un cas particulier, ou pour débugger le programme). Pour cela qu'il est utile lors de l'appel à `srand` d'afficher — ou d'envoyer dans un fichier de log — la graine choisie et de prévoir un moyen de la fixer, par un paramètre sur la ligne de commande ou une variable d'environnement.

4 Réduction de l'intervalle

Il est souvent nécessaire de ramener le nombre généré dans un intervalle plus réduit tout en conservant la répartition uniforme. Soit n la nouvelle valeur maximale désirée et M la valeur de `RAND_MAX`.

4.1 La bonne méthode

La première méthode qui vient à l'esprit est d'utiliser le reste d'une division par $n+1$:

```
int alea(int n)
{
  assert (0 < n && n <= RAND_MAX);
  return rand() % (n+1);
}
```

1. L'utilisation de ces sources est généralement trop coûteuse pour qu'elles remplacent les générateurs de nombres pseudo-aléatoires dans la plupart des applications.

Mais si $n+1$ n'est pas un diviseur de $M+1$ alors il y a un biais qui est introduit. En effet les nombres de 0 à $(M+1 \bmod n+1)-1$ ont une probabilité de

$$\frac{\lceil (M+1)/(n+1) \rceil}{M+1}$$

d'être générés tandis que les autres ont une probabilité de

$$\frac{\lfloor (M+1)/(n+1) \rfloor}{M+1}$$

La différence est d'autant plus importante que n est grand par rapport à M. Si on n'ignore pas certains nombres générés, n'importe quelle méthode va introduire un tel biais, seuls changeront les nombres favorisés. S'il faut éviter le biais, il faut donc parfois ignorer le résultat de `rand()` et recommencer l'appel :

```
int alea(int n)
{
  assert (0 < n && n <= RAND_MAX);
  int partSize = (RAND_MAX+1) / (n+1);
  int firstToBeDropped = partSize * (n+1);
  int draw;
  do {
    draw = rand();
  } while (draw >= firstToBeDropped);
  return draw % (n + 1);
}
```

Les générateurs par congruence linéaire – un type de générateurs de nombre pseudo-aléatoire courament utilisé – a un comportement qui rend l'utilisation du reste de la division pour réduire l'intervalle parfois peut désirable. En effet, pour certaines valeurs de leurs paramètres, la suite générée quand on la prend modulo k a pour certains k dépendants de ces paramètres une période beaucoup plus courte que la période du générateur, parfois simplement k. C'est génant quand notre $n+1$ est un des k problèmatiques ou un multiple de ceux-ci [2]. Il se fait que d'autres propriétés désirables des paramètres impliquent que les k problématiques sont les puissances de deux [3], ce qui rend vraissemblablement l'occurence du problème plus probable. Le problème est connu depuis longtemps, les bibliothèques évitent donc probablement ces valeurs des paramètres (ou même ce type de générateur) ou du moins utilisent des techniques pour corriger ce défaut [4]. Cependant, il n'est pas compliqué de l'éviter aussi dans le programme, il suffit de diviser par `partSize` plutôt que de prendre le modulo :

2. et non pour n'importe quelle valeur de $n+1$

3. On lit souvent – y compris dans les premières versions de ce document – que les bits de poids faibles sont moins aléatoires ; ce n'est vrai que pour ces paramètres. D'autres ont été utilisés qui avaient ce comportement indésirables pour les multiples d'autres petits nombres premiers.

4. Le générateur donné en exemple dans la norme C a des paramètres qui font que les k sont les puissances de 2 ; comme technique corrective il cache à l'appelant les 16 bits de poids les plus faible.

```
int alea(int n)
{
  assert (0 < n && n <= RAND_MAX);
  int partSize = (RAND_MAX+1) / (n+1);
  int firstToBeDropped = partSize * (n+1);
  int draw;
  do {
    draw = rand();
  } while (draw >= firstToBeDropped);
  return draw/partSize;
}
```

Nous en avons terminé avec nos efforts pour profiter au maximum de la qualité du générateur de la bibliothèque. Mais il reste quelques petits problèmes.

Si `RAND_MAX` est égal à `MAX_INT` [5], le calcul de `RAND_MAX + 1` a un dépassement de capacité. On peut l'éviter en soustrayant n de `RAND_MAX` avant la division et en additionnant un au résultat de celle-ci. Donc

```
int alea(int n){
  assert (0 < n && n <= RAND_MAX);
  int partSize = 1 + (RAND_MAX-n)/(n+1);
  int firstToBeDropped = partSize * (n+1);
  int draw;
  do {
    draw = rand();
  } while (draw >= firstToBeDropped);
  return draw/partSize;
}
```

Le calcul $n+1$ peut aussi faire un dépassement de capacité. Ce n'est possible que si $n+1$ vaut `MAX_INT` et donc `RAND_MAX`. Il faut aussi éviter le dépassement dans le calcul de `maxUsefull`. On a donc :

```
int alea(int n){
  assert (0 < n && n <= RAND_MAX);
  int partSize =
       n == RAND_MAX ? 1 : 1 + (RAND_MAX-n)/(n+1);
  int firstToBeDropped = partSize * n + partSize;
  int draw;
  do {
    draw = rand();
  } while (draw >= firstToBeDropped);
  return draw/partSize;
}
```

Si `RAND_MAX` vaut `MAX_INT` et que $n+1$ divise `RAND_MAX+1` alors `maxUsefull` n'est pas représentable puisqu'il vaut aussi `RAND_MAX+1`; changeons donc légèrement le test pour arriver à la version finale :

5. C'est qui est le cas pour l'implémentation que j'utilise le plus souvent.

```
int alea(int n){
  assert (0 < n && n <= RAND_MAX);
  int partSize =
       n == RAND_MAX ? 1 : 1 + (RAND_MAX-n)/(n+1);
  int maxUsefull = partSize * n + (partSize-1);
  int draw;
  do {
    draw = rand();
  } while (draw > maxUsefull);
  return draw/partSize;
}
```

4.2 Étude de quelques méthodes alternatives

Dans cette section, nous présentons un certain nombre de moyens qui ont été parfois présentés (jusque dans des FAQ) pour réduire l'intervalle et donnons leurs inconvénients. Par soucis d'homogénéité, ces méthodes sont présentées avec les notations de la section précédente (donc on cherche à obtenir un nombre entre 0 et n compris).

```
return rand() % (n+1);
```

C'est la version dont nous sommes partis ; pour rappel les problèmes que nous avons supprimés sont :

- la distribution est biaisée si $n+1$ n'est pas un diviseur de `RAND_MAX+1` (les valeurs favorisées sont groupées près de 0) ;
- on utilise le reste d'une division pour réduire l'intervalle
- il y a un dépassement de capacité possible.

C'est néanmoins la forme à utiliser si on sait que $n+1$ est représentable et qu'on accorde pas trop d'importance au problème du biais et au risque d'avoir une résonnance avec un mauvais paramètre d'un générateur à congruence linéaire.

```
return (int)((float)rand() / RAND_MAX * n);
```

- n n'est généré que quand `rand()` retourne `RAND_MAX` ;
- on perd une partie des bits aléatoires si `RAND_MAX` est plus grand que le plus grand entier pouvant être représenté sans perte dans un **`float`**, ce qui est parfois le cas (par exemple pour Linux) ;
- la distribution est biaisée si $n+1$ n'est pas un diviseur de `RAND_MAX+1` (les valeurs favorisées sont distribuées régulièrement dans tout l'intervalle).

```
return (int)((double)rand() / RAND_MAX * n);
```

- n n'est généré que quand `rand()` retourne `RAND_MAX` ;
- on perd une partie des bits aléatoires si `RAND_MAX` est plus grand que le plus grand entier pouvant être représenté sans perte dans un **`double`**, ce qui est peu vraisemblable (les **`double`** ayant généralement 53 bits de mantisse tandis que les **`int`** ont généralement 32 bits) ;

– la distribution est biaisée si $n+1$ n'est pas un diviseur de `RAND_MAX+1` (les valeurs favorisées distribuées régulièrement dans tout l'intervalle).

```
return (n+1) * rand() / (RAND_MAX + 1.0);
```

– il y a risque de dépassement de capacité si `RAND_MAX` vaut plus que la racine carrée de `MAX_INT` ;
– on perd une partie des bits aléatoires si `RAND_MAX` est plus grand que le plus grand entier pouvant être représenté sans perte dans un **`double`**, ce qui est peu vraisemblable (les **`double`** ayant généralement 53 bits de mantisse tandis que les **`int`** ont généralement 32 bits) ;
– la distribution est biaisée si $n+1$ n'est pas un diviseur de `RAND_MAX+1` (les valeurs favorisées distribuées régulièrement dans tout l'intervalle).

```
int maxUsefull = RAND_MAX - RAND_MAX % (n+1);
int partSize = maxUsefull / (n+1);
int draw;
do {
  draw = rnd();
} while (draw >= maxUsefull);
return draw / partSize;
```

– il y a risque de dépassement de capacité lors du calcul de `n+1` ;
– on n'utilise pas une partie de l'aléatoire disponible (par exemple on ne peut pas utiliser `n == RAND_MAX`).

5 Historique

26 mars 2006 création

6 avril 2006 orthographe, changements mineurs

19 novembre 2006 meilleure description du problème des générateurs à congruence linéaire

25 février 2009 utilisation de TEX4ht pour générer du HTML.

31 mars 2009 changements mineurs