Boost.Random : les nombres aléatoires de Boost

par Matthieu Brucher (http://matthieu-brucher.developpez.com/) (Blog)

Date de publication : 26/06/2006

Dernière mise à jour : 30/09/2006

La seule utilisation de la fonction rand() héritée du C est insuffisante pour la majorité des cas actuels. Il est en effet impossible de l'utiliser pour de la cryptographie, pour des distributions autres que l'uniforme sans devoir la programmer soi-même, ... C'est là que Boost.Random intervient.

Introduction

Proposée en partie pour le nouveau standard du C++, cette bibliothèque contient plusieurs générateurs aléatoires ainsi que plusieurs distributions à utiliser conjointement avec ces générateurs.

Selon l'objectif qu'on recherche, on choisira l'un ou l'autre des générateurs et la distribution qui va avec. Afin de simplifier l'écriture, une classe permettant de faire le lien entre le générateur et la distribution choisie a été ajoutée.

Cette bibliothèque est templatée et n'utilise pas de bibliothèque compilée .lib/.a ou .dll/.so. Toutes les classes et les fonctions vivent dans le namespace **boost::random**, sauf **rand48()** qui vit dans **boost**.

I - Les générateurs aléatoires uniformes

Boost propose une quantité impressionnante de générateurs uniformes, de types variés, mais aussi avec des capacités différentes, adaptés à des problèmes précis. Chaque générateur produit un nouveau nombre aléatoire en appelant **operator()()**, ce sont donc des foncteurs. Chaque générateur peut être initialisé avec un appel à la fonction **seed(...)**, chaque classe définit aussi **min_value** et **max_value**, les valeurs minimales et maximales en sortie et un booléen **has_fixed_range** indiquant si l'itervalle des nombres générés est fixe ou pas. Voici un rapide synopsis et un tableau des générateurs utilisables.

Outre une émulation de la fonction C **lrand48()** appelée **rand48()**, une classe de générateurs selon une congruence linéaire, une classe de générateurs utilisant les nombres de Mersenne ainsi qu'une classe utilisant un algorithme de Fibonacci attardé sont disponible, en plus de quelques spécialités.

Générateur	Longueur du cycle	Taille approximative en mémoire	Vitesse approximative
minstd_rand	$2^{31}-2$	sizeof(int32_t)	40
rand48	$2^{48}-1$	sizeof(int64_t)	80
lrand48 (C library)	$2^{48}-1$	-	20
ecuyer1988	environ $2^{61}-1$	2*sizeof(int32_t)	20
kreutzer1986	?	1368*sizeof(uint32_t)	60
hellekalek1995	$2^{31}-1$	sizeof(int32_t)	3
mt11213b	$2^{11213}-1$	352*sizeof(uint32_t)	100
mt19937	$2^{19937}-1$	625*sizeof(uint32_t)	100
lagged_fibonacci607	environ 2^{32000}	607*sizeof(double)	150
lagged_fibonacci1279	environ 2^{67000}	1279*sizeof(double)	150
lagged_fibonacci2281	environ 2^{120000}	2281*sizeof(double)	150
lagged_fibonacci3217	environ 2^{170000}	3217*sizeof(double)	150
lagged_fibonacci4423	environ 2^{230000}	4423*sizeof(double)	150
lagged_fibonacci9689	environ 2^{510000}	9689*sizeof(double)	150
lagged_fibonacci19937	environ $2^{1050000}$	19937*sizeof(double)	150
lagged_fibonacci23209	environ $2^{1200000}$	23209*sizeof(double)	140

	environ $2^{1200000}$		
lagged_fibonacci	random_size_t	sizeof(double)	60
	$2^{2300000}$		

I-A - Les générateurs à congruence linéaire

Ici, un nouveau nombre aléatoire est généré en fonction du précédent à l'aide d'une fonction congruente :

```
x(n+1) = (a * x(n) + c) mod m
```

Les paramètres templates de la classe sont le type d'entier utilisé, les facteurs **a**, **c** et **m** ainsi que la valeur **val** qui sera testée lors de l'appel à la fonction **validation()**. Il s'agit en fait du 10001ème élément généré en partant du constructeur par défaut.

Pour simplifier l'utilisation de la classe, 2 **typedef** sont proposés :

```
typedef random::linear_congruential<long, 16807L, 0, 2147483647L, 1043618065L> minstd_rand0;
typedef random::linear_congruential<long, 48271L, 0, 2147483647L, 399268537L> minstd_rand;
```

L'en-tête correspondant est **boost/random/linear_congruential.hpp**.

I-B - Les générateurs Mersenne Twister

Le cycle d'un générateur par congruence est limité au modulo qu'on lui impose. Lorsque le nombre est premier, le cycle peut être de même longueur que le modulo. Les générateurs Mersenne Twister utilisent les nombres de Mersenne qui sont des nombres premiers de la forme 2^n - 1. Outre la congruence linéaire, le générateur utilise une équation supplémentaire pour rendre plus aléatoire les nombres, le twister. En effet, sans cette modification, le générateur ne serait qu'un générateur par congruence linéaire.

L'inconvénient des générateurs MT est la taille de l'état qui doit être stocké, puisqu'il y a un cycle de taille 2^n -1, il faut savoir où l'on est. De plus, l'équation de mélange est complexe et nécessite de savoir ce que l'on fait. Pour répondre à cette problématique, 2 exemples de générateurs MT sont fournis, le plus connu étant le deuxième. Par défaut, utilisez-le, il est rapide et robuste.

```
typedef mersenne_twister<uint32_t,351,175,19,0xccab8ee7,11,7,0x31b6ab00,15,0xffe50000,17, /*
 unknown */ 0> mt11213b;
typedef mersenne_twister<uint32_t,624,397,31,0x9908b0df,11,7,0x9d2c5680,15,0xefc60000,18,
 3346425566U> mt19937;
```

L'en-tête correspondant est **boost/random/mersenne_twister.hpp**.

I-C - Les générateurs avec Fibonacci attardé

Les générateurs de nombres aléatoires basés sur ce principe sont des générateurs de nombres flottants entre 0. et 1. L'équation de génération est dérivée de l'équation de Fibonacci :

```
x(i) = x(i-p) + x(i-q) (mod 1)
```

Il faut donc retenir un nombre de valeurs générées important, d'autant plus que q et p sont grands. Pour cela, plusieurs typedefs sont aussi fournis :

```
typedef random::lagged_fibonacci<double, 607, 273> lagged_fibonacci607;
typedef random::lagged_fibonacci<double, 1279, 418> lagged_fibonacci1279;
typedef random::lagged_fibonacci<double, 2281, 1252> lagged_fibonacci2281;
typedef random::lagged_fibonacci<double, 3217, 576> lagged_fibonacci3217;
typedef random::lagged_fibonacci<double, 4423, 2098> lagged_fibonacci4423;
typedef random::lagged_fibonacci<double, 9689, 5502> lagged_fibonacci9689;
typedef random::lagged_fibonacci<double, 19937, 9842> lagged_fibonacci19937;
typedef random::lagged_fibonacci<double, 23209, 13470> lagged_fibonacci23209;
typedef random::lagged_fibonacci<double, 44497, 21034> lagged_fibonacci44497;
```

L'en-tête correspondant est **boost/random/lagged_fibonacci.hpp**.

I-D - Les autres générateurs

D'autres générateurs sont disponibles, plus complexes.

Le premier est un typedef d'une classe qui fait une addition de 2 générateurs à congruence, il s'agit de **ecuyer1988** qui est la combinaison de **linear_congruential<int32_t, 40014, 0, 2147483563, 0>** et de **linear_congruential<,int32_t, 40692, 0, 2147483399, 0>**.

Le second, **kreutzer1986**, est aussi basé sur un générateur par congruence, mais en modifiant l'ordre des valeurs générées. En effet, on choisit une case dans un tableau de taille fixe, ici **97**, puis on la remet à jour, et le choix de la prochaine case sera fonction aussi de la valeur précédente générée. Le générateur par congruence utilisé ici est **linear_congruential<uint32_t, 1366, 150889, 714025, 0>**.

Le dernier est un générateur à congruence inverse, on ne rentrera pas dans les détails quand à son fonctionnement.

```
typedef random::inversive_congruential<int32_t, 9102, 2147483647-36884165, 2147483647>
 hellekalek1995;
```

Les en-têtes correspondants sont **boost/random/additive_combine.hpp**, **boost/random/inversive_congruential.hpp** et **boost/random/shuffle_output.hpp**.

II - Les distributions

Les distributions les plus courantes sont proposées par Boost.Random. Pour générer un nombre aléatoire selon une certaine distribution, il faut avoir au préalable un générateur de nombres aléatoires uniforme, et l'appel se fera sous cette forme :

```
distribution(generateur);
```

II-A - Les distributions entières uniformes

Ces distributions retournent un entier - par défaut de type **int** - dans un intervale connu. **uniform_smallint<>** prend en outre comme hypothèse que l'intervale dans lequel le nombre sera tiré est petit devant l'intervale dans lequel le générateur aléatoire tire une valeur. En revanche, **uniform_int<>** n'utilise pas cette hypothèse et tire donc correctement un entier. La petite différence n'est pas critique lorsque l'hypothèse est vérifiée - c'est la différence entre utiliser **rand() % 20** et **rand() / 20**.

Les constructeurs prennent en argument la valeur minimale et la valeur maximale de l'intervale. A noter que **uniform_int<>** peut être appelée par **operator()** pour générer un nombre dans l'intervalle **[0; n]**.

Les en-têtes correspondants sont **boost/random/uniform_smallint.hpp** et **boost/random/uniform_int.hpp**.

II-B - Les distributions réelles uniformes

Ici, on a accès à la classique distribution dans l'intervalle **[0; 1[** représentée par **uniform_01<>** et à celle sur l'intervalle **[min; max[**, **uniform_real<>**.

Les constructeurs prennent en argument la valeur minimale et la valeur maximale de l'intervalle.

Les en-têtes correspondants sont **boost/random/uniform_01.hpp** et **boost/random/uniform_real.hpp**.

II-C - Les autres distributions réelles

La première classe dans cette catégorie est le tirage de Bernouilli. **bernoulli_distribution<>** effectue un tirage de Bernouilli, sachant qu'on donne la probabilité d'apparition du **true** dans le constructeur.

La distribution géométrique **geometric_distribution<>** est de la même famille que **bernoulli_distribution<>**. Elle produit des entiers naturels **i** non nuls avec la probabilité :

```
p(i) = (1-p) * p^(i-1)
```

La distribution triangulaire **triangle_distribution<>** tire des nombres dans un intervalle donné en paramètre, l'élément le plus probable étant lui aussi donné en paramètre.

La distribution exponentielle **exponential_distribution<>** génère des nombres selon une exponentielle inverse dont l'argument est donné à la construction de la distribution.

La distribution normale, ou gaussienne, **normal_distribution<>** génère des nombres selon une gaussienne dont les arguments sont donnés à la construction de la distribution.

La distribution log-normale, **lognormal_distribution<>** génère des nombres selon une loi log-normale dont les arguments sont donnés à la construction de la distribution.

La distribution uniforme sur une sphère, **uniform_on_sphere<>** génère des nombres uniformément sur une sphère dont la dimension est donnée à la construction de la distribution.

Les en-têtes correspondants sont **boost/random/triangle_distribution.hpp**, **boost/random/bernoulli_distribution.hpp**, **boost/random/cauchy_distribution.hpp**, **boost/random/exponential_distribution.hpp**, **boost/random/geometric_distribution.hpp**, **boost/random/normal_distribution.hpp**, **boost/random/lognormal_distribution.hpp** ou encore **boost/random/uniform_on_sphere.hpp**.

III - La glue entre générateurs et distributions

L'objet **variate_generator<>** permet d'associer un générateur et une distribution. Leur type est donné en paramètre template, ce qui permet de choisir entre une copie ou une référence vers le générateur, afin d'utiliser le même générateur pour toutes les générations et non pas plusieurs copies du même, ce qui ne génèrerait pas des nombres réellement aléatoires. Cette classe est un foncteur, tout comme les générateur de nombres aléatoires.

Exemple d'utilisation

```
#include <boost/random/mersenne_twister.hpp>
#include <boost/random/normal_distribution.hpp>
#include <boost/random/variate_generator.hpp>
#include <ctime>

void test()
{
    boost::mt19937 engine(static_cast<long unsigned int>(clock()));
    boost::normal_distribution<double> generator;
    boost::variate_generator<boost::mt19937, boost::normal_distribution<double> > binded(engine,
 generator);
}
```

IV - Quelques objets annexes

Il existe des générateurs non-déterministes ainsi que des décorateurs qui n'ont pas encore été présentés. Le générateur non déterministe n'est malheureusement pas présent sur toutes les plateformes car son implémentation ne peut être effectuée en C++. En ce qui concerne les décorateurs, il s'agit à nouveau d'un singulier qui permet d'utiliser les générateurs aléatoires comme des générateurs sur un intervalle **[O, n[** où **n** est donné lors de l'appel à la méthode **()**, ce que la classe **uniform_in<>** permet presque de faire.

Conclusion

Les algorithmes utilisés par cette bibliothèque sont standards et éprouvés. D'ailleurs si vous cherchez à implémenter un générateur de nombres aléatoires avec une distribution gaussienne, vous réécrirez exactement le code de Boost, pas très intéressant. De plus, la future disponibilité de ces classes dans la bibliothèque standard du C++ encourage aussi à se jeter à l'eau et à les utiliser.

Un dernier point à surveiller, les générateurs uniformes ont principalement des états internes. On peut les utiliser pour remplir un tableau avec **std::generate**, mais si vous faites cela plusieurs fois d'affilée, vous remplirez vos conteneurs avec les mêmes valeurs. C'est pourquoi, lors de l'utilisation de **boost::random::variate_generator**, n'oubliez pas le **&** pour indiquer l'utilisation d'une référence vers le générateur aléatoire global.