Les nombres pseudo-aléatoires

Un nombre aléatoire est un nombre dont la valeur est choisit au hasard. Ils sont utilisé dans de nombreux contextes, par exemple dans un jeu vidéo (pour avoir un comportement non prédictible), en cryptographie (pour encoder des informations), dans les tests automatisés du code (pour vérifier le comportement d'un programme avec des valeurs indéterminées). Vous verrez dans différents projets de ce cours des cas d'utilisation des nombres aléatoires.

Qualité des nombres aléatoires

Tirer des nombres aléatoires peut sembler être une tâche facile. On peut par exemple prendre un dé de jeux (non pipé) à six faces et le lancer plusieurs fois pour obtenir une série de nombres aléatoires compris entre un et six. On peut également lancer une pièce de monnaie pour obtenir des nombres aléatoires valant zéro (Pile) ou un (Face).

La difficulté vient du fait qu'il est facile d'obtenir des suites de nombres dont on n'arrive pas à prédire la prochaine valeur, mais qui présente quand même des propriétés mathématiques prédictibles. Si on additionne par exemple la valeur de deux dés, on aura un nombre aléatoire compris entre 2 et 12. Mais la probabilité d'obtenir un 12 est bien inférieure (1 chance sur 36) à la probabilité d'obtenir un 7 (12 chances sur 36).

Quand vous jouez à un jeu de société, ce n'est pas très important que les nombres obtenus ne soient pas parfaitement aléatoire (et c'est de toute façon intégré comme composante du jeu). Mais lorsque les nombres aléatoires permettent de crypter des données importantes ou sécuriser une numéro de carte bancaire, la moindre propriété mathématique prédictible sur une série de nombres aléatoires peut permettre de contourner ces sécurités.

Il ne faut donc pas simplement que les nombres semblent aléatoires. Il faut le prouver mathématiquement. (Et bien sûr, cela sort complètement du domaine de ce livre, ces sont des travaux de recherches très pointus).

Pour en revenir à la programmation, on peut se poser une question : comment générer des nombres parfaitement aléatoires sur un ordinateur ? Ce n'est pas si simple, puisqu'un ordinateur est prévu pour être parfaitement prédictible (la même suite d'instructions avec les mêmes données produira le même résultat).

Heureusement, il existe plusieurs approches possibles.

Nombres aléatoires en C++

La bibliothèque standard fournit plusieurs fonctionnalités pour générer des nombres aléatoires, en utilisant différents algorithmes. L'ensemble de ces fonctionnalités sont accessibles en utilisant le fichier d'en-tête `random` (“aléatoire” en anglais).

```
#include <random>
```

La documentation est accessible en ligne sur cppreference.comm: Pseudo-random number generation, n'hésitez pas à vous familiariser un peu avec elle avant de continuer ce chapitre.

Les fonctionnalités de génération de nombres aléatoires se décomposent en trois catégories :

- le générateur de nombres aléatoires non déterministe ;
- les générateurs de nombres aléatoires déterministes (nombres pseudo-aléatoires) ;
- les générateurs de distributions statistiques.

Si vous travaillez sur du code qui a besoin d'un générateur le plus aléatoire possible, il faudra vérifier le fonctionnement du générateur ou utiliser une bibliothèque dédiée à la génération de nombres aléatoires pour la cryptographique (ces générateurs sont plus lents que les générateurs utilisés dans la bibliothèque standard, ce qui explique

pourquoi ils ne sont pas utilisés par défaut).

Générateurs non déterministes

L'utilisation d'un générateur aléatoire est relativement simple, même si cela utilise une syntaxe que vous n'avez pas encore vu. Il faut créer une variable utilisant le générateur comme type, puis appeler cette variable comme une fonction. On parle d'"objet appelable" (*callable* en anglais). En pratique, la syntaxe est la suivante :

```
CallableType myVariable{};              // créé une variable
de type CallableType
auto result = myVariable(arg1, arg2);   // appel cette
variable comme une fonction
```

Le type `CallableType` représente n'importe quel type qui permet de créer un objet appelable. Comme n'importe quelle fonction, un objet appelable peut prendre des arguments en entrée et retourner une valeur.

Le générateur non déterministe est la classe `std::random_device`. Cet objet sera appelé sans utiliser d'argument, et retourne un entier aléatoire à chaque fois qu'il est appelé.

main.cpp

```
#include <iostream>
#include <random>

int main() {
    std::random_device rd{};            // création du
générateur

    std::cout << rd() << std::endl;     // génération d'un
nombre aléatoire
    std::cout << rd() << std::endl;     // génération d'un
nombre aléatoire
    std::cout << rd() << std::endl;     // génération d'un
nombre aléatoire
}
```

Ce code permet de générer trois nombres aléatoires, qui seront différents à chaque fois que vous relancez le programme. Par exemple, cela peut donner la série suivante :

```
3555021123
4089866385
3885819577
```

En pratique, l'implémentation des générateurs n'est pas définie dans la norme C++, chaque implémentation de la bibliothèque standard peut avoir un comportement différent. Il n'est donc pas possible de prévoir le comportement réel de ce générateur.
Par exemple, sur Coliru, à chaque fois que le programme est relancé, il va générer la même suite de nombres aléatoires. Le même code sur Ideone va produire des nombres aléatoires différents à chaque fois que le programme est lancé.

Pour savoir si le générateur non déterministe est effectivement non déterministe ou pas, il faut afficher l'entropie en utilisant la fonction `entropy` :

main.cpp

```cpp
#include <iostream>
#include <random>

int main() {
    std::random_device rd;
    std::cout << "Entropy: " << rd.entropy() << std::endl;
}
```

Si la valeur affichée est nulle, alors le générateur est déterministe.

Le problème de ce générateur est qu'il n'est pas possible de prouver qu'il a un comportement parfaitement aléatoire, qu'on ne peut prédire aucune caractéristique mathématique sur la série de nombres générés. On préférera donc généralement les générateurs de nombres aléatoires déterministes.

Générateurs déterministes

Un générateur déterministe produit des nombres aléatoires selon un algorithme complexe donné. Ce type de générateur prend en argument une graine (*seed* en anglais), qui définie la série de nombres aléatoires à générer. Utiliser deux fois la même graine produit la même série de nombres.

Graine d'un générateur

Dans de nombreux cas, on ne souhaite pas utiliser une graine en particulier, on voudra avoir une série différente de nombres aléatoires, à chaque fois qu'on lance une application. Par exemple :

- dans un jeu, pour que les monstres aient un comportement différent à chaque partie, sans que l'on puisse prédire ce qu'ils vont faire ;
- dans un programme testant la qualité d'un code (test unitaire), pour avoir des valeurs qui changent à chaque fois que l'on relance les tests ;
- dans un programme d'encodage de données, pour que l'on ne puisse pas prédire l'encodage utilisé.

Mais dans certains cas, il sera intéressant d'utiliser une graine connue. Un exemple classique est un programme de test de code, qui détecte une erreur en utilisant une série spécifique de nombres aléatoires. Pour reproduire l'erreur, on pourra générer la même série de nombres en utilisant la même graine.

Pour créer un générateur en utilisant une graine, il suffit de mettre cette graine comme argument lors de la création du générateur. (Note : il existe plusieurs générateurs déterministes, qui seront vu par la suite. Pour le début de ce cours, nous allons utiliser le générateur `std::default_random_engine`).

```
std::default_random_engine engin { seed };
```

Ce code créé une variable nommée `engin`, de type `std::default_random_engine` et initialisé avec la valeur `seed`.

Pour générer les nombres aléatoires, il faut ensuite appeler la variable `engin` comme une fonction, comme vous avez fait pour `random_device` :

main.cpp
```
#include <iostream>
#include <random>

int main() {
    std::default_random_engine engin { 123 };  // création du générateur

    std::cout << engin() << std::endl;         // génération d'un nombre aléatoire
    std::cout << engin() << std::endl;         // génération d'un nombre aléatoire
    std::cout << engin() << std::endl;         // génération d'un nombre aléatoire
}
```

affichera :

```
2067261
384717275
2017463455
```

Remarque : cette série de nombres est déterministe, ce qui veut dire que si vous utiliser le même générateur et la même graine que dans le code, vous devriez avoir exactement la même série de nombres aléatoires (contrairement au `random_device`, pour lequel vous pouvez avoir un résultat différent que celui donné dans le cours).

Pour générer une série en utilisant une graine qui change a chaque fois que le programme est lancé, il est classique d'initialiser le générateur avec le temps (ce qui garantie que la graine change à chaque fois). Par exemple :

main.cpp

```cpp
#include <iostream>
#include <random>
#include <chrono>

int main() {
    const auto seed = std::time(nullptr);
    std::cout << seed << std::endl;
    std::default_random_engine engin { seed };
}
```

La fonction `std::time` retourne le temps passé depuis une date fixée (généralement 1970), en secondes.

Encore une fois, on peut remarquer des “failles” dans le système de génération des nombres aléatoires. Du fait que la graine change toutes les secondes, cela veut dire que si on génère deux graines trop rapidement, elles seront identiques.
De même, si on sait à peut prêt quand un nombre aléatoire a été généré, on peut retrouver plus facilement la graine qui a été utilisée et la série de nombres aléatoires.

Dans un programme critique, par exemple en programme d'encodage de données importantes, cela constituent des failles de sécurité. On voit ici qu'il est finalement assez facile, si on ne fait pas attention, de rendre prédictible un générateur et donc de créer des failles de sécurité dans un programme.

L'utilisation d'un générateur aléatoire de qualité n'est pas suffisant, il faut aussi l'utiliser correctement.

Pour des applications qui nécessite un niveau de sécurité élevé, on utilisera à la place des générateurs dédiés à la cryptographie.

Les différents générateurs

Plusieurs générateurs déterministes sont proposés dans la bibliothèque standard du C++. Ces générateurs sont définies par l'algorithme utilisé et par les valeurs utilisées pour le configurer. Pour comprendre cela, regardons un peu plus en détail la documentation : Pseudo-random number generation.

Trois algorithmes de génération sont utilisé. La documentation donne des liens vers les pages de Wikipédia décrivant ces algorithmes.

- `std::linear_congruential_engine` : algorithme congruentiel linéaire ;
- `std::mersenne_twister_engine` : algorithme de Mersenne Twister ;
- `std::subtract_with_carry_engine` : algorithme de soustraction avec retenu.

Il est très probable que vous ne connaissez pas ces algorithmes... et cela ne va pas changer tout de suite :) Le but de ce cours est de vous expliquer comment utiliser les générateurs aléatoires et lire la documentation, pas d'expliquer chaque algorithme en détail. Pour cela, il vous faudra lire les documentations, voire lire un cours de cryptographie.
Il existe ensuite trois adaptateurs, qui permettent de modifier un autre algorithme. Les adaptateurs de la bibliothèques standard sont :

- `std::discard_block_engine` ;
- `std::independent_bits_engine` ;
- `std::shuffle_order_engine`.

Et pour terminer, les différents générateurs proposés par défaut, à partir de ces algorithmes et adaptateurs :

- `std::minstd_rand0` ;
- `std::minstd_rand` ;

- `std::mt19937` ;
- `std::mt19937_64` ;
- `std::ranlux24_base` ;
- `std::ranlux48_base` ;
- `std::ranlux24` ;
- `std::ranlux48` ;
- `std::knuth_b`.

Ces générateurs s'utilisent de la même façon que le générateur `std::default_random_engine` cité précédemment. Par exemple, pour `std::ranlux48_base`, avec une graine fixe :

main.cpp

```
#include <iostream>
#include <random>

int main() {
    std::ranlux48_base engin { 123 };          // création
du générateur

    std::cout << engin() << std::endl;         // génération
d'un nombre aléatoire
    std::cout << engin() << std::endl;         // génération
d'un nombre aléatoire
    std::cout << engin() << std::endl;         // génération
d'un nombre aléatoire
}
```

affiche :

```
52617863149155
181669238551779
228994550894105
```

Comment générer un nombre aléatoire en pratique ?

Au final, cela fait beaucoup d'algorithmes à utiliser, vous ne savez probablement pas quel algorithme utiliser parmi tous les générateurs proposés. La réponse est simple :

- si cela n'a pas d'importance, uniquement `std::default_random_engine` ;
- si c'est important : uniquement les algorithmes que vous avez étudié et que vous comprenez.

Générateurs de distributions statistiques

Générateurs précédents : chaque nombre à même chance d'etre générer. Mais souvent besoin de nombres aléatoires qui suivent une loi particulière. Par exemple, somme de deux dés 6 non equiprobable, mais suis loi particulière. Générateur distributions permettent de générer ces lois

HS sujet, voir cours de stat. Quelques lois importantes : - uniforme entre min et max : équiprobable, dans un range. Version int et real - binomiale : pile ou face - loi normale

En pratique

Le plus souvent, loi uniforme, générateur par défaut, graine aléatoire.

Travaux pratiques

Donner une série de nombres aléatoires, produit avec un algo (donné ou non), une graine comprise entre 0 et 100. Essayer de retrouver la graine/algo

Chapitre précédent	Sommaire principal	Chapitre suivant

Cours, C++