

乱数制御

－ プレイヤーに不満を感じさせないようにするために －

2014 年 1 月 20 日 初版

板垣 衛

■ 改訂履歴

版	リリース	担当	改訂内容
初版	2014 年 1 月 20 日	板垣 衛	(初版)

■ 目次

■ 概略	1
■ 目的	1
■ 対処すべき乱数の問題	1
■ 知っておくべき乱数の性質	1
▼ 擬似乱数	1
▼ 乱数生成器	2
▼ 乱数列の周期	2
▼ 乱数の分布	2
▼ 乱数の生成順序	3
▼ アルゴリズム	3
● 線形合同法	3
● メルセンヌ・ツイスター	3
■ 様々な乱数 Tips	4
▼ 【重要】個人の判断で rand() 関数を使わない	4
▼ 乱数の小数化で「均等分布」性を常に反映	4
▼ 複数の乱数生成器の活用	5
● 乱数生成器の種類と使い分け	5
▼ シードの活用①：乱数のパターン化を避ける	6
▼ シードの活用②：乱数を再現する	7
● シード書き換えの注意点	7
▼ もっと乱数のランダム性を強化	7
▼ 乱数生成器を自作	7
▼ C++11 の乱数	8

■ 概略

乱数をゲームで使用する際に気を付けるべき点について解説。

特に、プレイヤーが乱数を強く意識するような要素の対処方法に重点を置く。

乱数は、チーム開発では意外と気を使わなければならない要素であることから、初歩的な内容を含めて解説する。

■ 目的

本書は、一人一人のプログラマーが乱数の性質を理解して、チーム開発の中で問題なく効果的に乱数を使用することを目的とする。

個人の判断で乱数を使用していると、チーム開発では思いがけない問題を招くことがある。

■ 対処すべき乱数の問題

まず、ゲームでありがちな乱数の問題を示す。

- ・ レアアイテムのドロップ率が、狙っている確率よりもずっと低い。
- ・ 当たり→はずれ→当たり→はずれが交互に繰り返され、単調な挙動になる。
- ・ 1/2 の確率で生き返るはずが、10 回も連続で失敗することがある。
- ・ ゲームを始めるたびに、毎回敵が同じ行動を取る。
- ・ 通信プレイで、プレイヤーごとに挙動が異なる。

■ 知っておくべき乱数の性質

▼ 擬似乱数

まず、当然のことながら、コンピュータの乱数は全くの無作為に選出されるような値ではなく、秩序ある計算に基づいて、「乱数に見える値」が算出される仕組みである。このような乱数を「擬似乱数」と呼ぶ。

このため、条件（乱数の計算の元になる係数）さえ合わせれば、いつどこで乱数を計算しても、必ず同じ値となる。

▼ 乱数生成器

乱数を生成するアルゴリズム（を実装したプログラム）を指して、一般に「乱数生成器」もしくは「擬似乱数生成器」と呼ばれる。「乱数ジェネレータ」「乱数エンジン」などの呼び方も同義。

「乱数生成器」には様々なものがあり、「良質な乱数」の生成を求めて採択される。

「良質な乱数」には、乱数の分布が十分であることのほか、算出される乱数の範囲（大きさ）、計算の速度、計算に使われるバッファ（メモリ）の大きさなどを考慮する必要がある。

▼ 乱数列の周期

前述の通り、乱数は条件が合えば同じ結果を算出する。

乱数を算出し続けると、いずれは最初の条件と同じ状態になり、また同じ乱数の算出が続く繰り返しとなる。これを「乱数の周期」と呼ぶ。

例：

8 → 0 → 2 → 3 → 8 → 0 → 2 → 3 → 8 → 0 → 2 → 3 → ...（繰り返し）

▼ 乱数の分布

C 言語標準ライブラリの `rand()`関数が生成する乱数の範囲は 0 ～ `RAND_MAX` の値である。`RAND_MAX` は通常 `0x7fff` ($32,767 = 16$ ビットの正の数の最大値) である。

まず、この範囲で十分なのかどうか、その乱数生成器を使ってよいかどうかの判断基準となる。もし、0～100 万の乱数が必要なら不十分である。

また、重要なのは、`rand()`関数を実行するたびに、十分にばらけた値が返されるかどうかである。最小値～中央値～最大値のそれぞれの値が、確実に均等に出現する「均等分布」が「良質な乱数」の条件である。

なお、「0～`RAND_MAX`」の値の中には、「一度も出現しない値もある」という点に注意。

▼ 乱数の生成順序

標準関数といえど、環境（コンパイラのメーカーなど）によっては、偶数→奇数→偶数→奇数の繰り返しとなったり、小さい→大きい→小さい→大きいが単調に繰り返されたりといった、分かり易い結果になることがある。

このような乱数の性質は、できるだけ事前に把握しておくべきである。この生成順序の問題は、ある程度使う側の工夫で対処できる。

▼ アルゴリズム

よく使われる乱数生成器を一部簡単に紹介する。

● 線形合同法

最も一般的な計算方法。C 言語の標準ライブラリでも使用されている。特徴は下記の通り。

- 計算に使用されるワーキングメモリが小さい（32bit 版で 4 バイト）
- 計算が単純で速い
- C 言語標準（32bit 版）だと値の範囲が狭い（ $0 \sim 2^{15} - 1 = 32,767$ ）
- 周期が短い（といっても 32bit 環境では最大で $2^{31} - 1 =$ 約 20 億はある）
- その他、「多次元で均等分布しない：(x1, y1) (x2, y2),...と順に乱数で求めると規則性が生じる」「下位ビットのランダム性が低い」などの多数の問題点が指摘されている

● メルセンヌ・ツイスター

Boost C++ や C++11 でも採用されている、比較的新しい乱数生成器。特徴は下記の通り。

- 計算が速い。
- 周期が長い（ $2^{19937} - 1$ ）
- 高次元（623 次元）に均等分布
- 出力の中のすべてのビットが統計的に十分にランダム
- 計算に使用されるワーキングメモリが大きい（32 版で 4992 バイト）
- 初期状態空間（ワーキングメモリの初期状態）に 0 が多いと、算出される乱数は、しばらくの間 0 が多くなる。

■ 様々な乱数 Tips

▼ 【重要】個人の判断で rand() 関数を使わない

先に説明したように、乱数は問題を起こしがちな要素で、その扱いは非常にデリケート。
開発プロジェクトごとに、使用する乱数生成器とその使い方を規定すべきであり、乱数の使用箇所は完全に把握しておくべきである。

このため、個人の判断で rand()関数を気軽に使ったりせず、開発プロジェクトの方針に従うことが重要である。

▼ 乱数の小数化で「均等分布」性を常に反映

乱数に対する要件は「OK か NG か?」「0~5 の乱数」「10~20 の乱数」といった形が通常である。

「rand() / RAND_MAX * 求めたい乱数の範囲」という計算を行うと、乱数生成器の均等分布性を常に結果に反映されることができる。

以下、二種類の計算方法の比較を示す。

例：乱数を割った余りで求める方法

```
//1 万回実行
int r = rand() % 5; //0~4 の乱数生成
```

↓ (結果)

```
[0] = 1997 回
[1] = 1951 回
[2] = 1985 回
[3] = 1965 回
[4] = 2102 回
最小: [1] = 1951 回
最大: [4] = 2102 回
最小と最大の差 = 151 回
```

例：乱数を小数化してから求める方法

```
//1 万回実行
double ratio = static_cast<double>(rand() / static_cast<double>(RAND_MAX));
int r = static_cast<int>(ratio * static_cast<double>(5)); //0~4 の乱数生成
```

↓ (結果)

```
[0] = 2029 回
[1] = 2022 回
[2] = 1954 回
[3] = 2015 回
[4] = 1979 回
最小: [2] = 1954 回
最大: [0] = 2029 回
最小と最大の差 = 75 回
```

▼ 複数の乱数生成器の活用

「1/2 の確率で仲間が生き返る魔法」というようなものは、プレイヤーがその「確率」を強く意識する。これが「10 回も連続で失敗」というようなことがあると、プレイヤーはゲームに不信感を持ってしまう。

このような問題が起こるのは「乱数生成器」の性能の問題だけではない。

例えば、ゲームで使用している乱数生成器が「小さい値 → 大きい値 → 小さい値 → 大きい値 → ...」と繰り返す傾向が強いとする。

それに対して、ある場面で乱数を使用される順序が「生き返り魔法成功率計算 → 敵の攻撃ダメージ計算 → 生き返り魔法成功率計算 → 敵の攻撃ダメージ計算 → ...」という状況になった時、「生き返り魔法成功率計算」の時に返される乱数が常に「小さい値」になってしまい、結果として失敗を続けてしまうという状況に陥る。

このような問題の対処としては、一つの乱数生成器 (rand()関数など) で全ての乱数を生成することを考え直し、特にプレイヤーが確率を意識するような要素に対しては、専用の乱数生成器を使用するような方法が効果的である。

例：乱数生成器の使い分け

```
//ダメージ計算
int damage = base + normal_rand(5); //base ダメージ値 + 0~4 の乱数

//生き返り成功率
bool is_revived = (revive_rand(100) < 50); //生き返り成功率専用乱数生成器
```

● 乱数生成器の種類と使い分け

では、どれだけの数の乱数生成器を用意すべきか？

単純にな区分けとしては次の三つである。

➤ 通信プレイやリプレイ時に使用される乱数

後述する「乱数の再現性」の意識が必要になる場面がある。

通信プレイやリプレイでは、少ない情報量で正確な共通性・再現性を実現するために、乱数を一致させることで済ませる手法がある。

➤ 通信プレイやリプレイ時に使用され、プレイヤーが強く確率を意識する要素

➤ 通信プレイやリプレイ時に使用されない乱数

例えば「草木の揺れ方」などは通信プレイやリプレイで多少違っていてもまず問題がない。このような「ゲームプレイに影響がないが乱数が使いたい箇所」に適用するための乱数生成器を用意しておく、問題を起こしにくい。

このような要素と乱数生成器を共有していると、時には「処理落ちしたプレイヤーだけ挙動が違

う」といった追跡の難しいバグを生むことがあるので、あらかじめ気をつけておくべき。

また、「再現性」が求められる乱数の生成は、「処理落ち」などのタイミングに影響しない形で乱数を生成する、ということも重要。

▼ シードの活用①：乱数のパターン化を避ける

乱数のテストプログラムなどを作ると顕著に分かるが、プログラムを実行するたびに毎回同じ順列の乱数が出現する。

これは、「シード」の初期値が一定のためである。

「シード(種)」は、乱数生成の元になる係数である。このシードを任意に変えることで、乱数の初期値を変えることができる。

プログラムを実行するたびに乱数の結果を変えたい場合は、現在時間などを使用してシードを書き換えるのが単純な方法である。

例：現在時間でシードを書き換え

```
#include <time.h>

//現在時間でシードを書き換え
srand(static_cast<unsigned int>(time(nullptr)));
printf("rand()=");
for (int i = 0; i < 10; ++i) printf(" %d", rand());
printf("\n");
```

↓ (結果：1回目)

```
rand()= 14809 10413 8548 5526 21545 25704 14200 15312 23393 18259
```

↓ (結果：2回目)

```
rand()= 15054 30112 4873 8035 8706 25297 18258 25142 16578 15906
```

↓ (結果：3回目)

```
rand()= 15149 14135 31414 17743 12917 8100 10215 5787 3894 1452
```

例：シードを書き換えなかった場合

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

↓ (結果：1回目)

```
rand()= 41 18467 6334 26500 19169 15724 11478 29358 26962 24464
```

↓ (結果：2回目) ※1回目と同じ

```
rand()= 41 18467 6334 26500 19169 15724 11478 29358 26962 24464
```

↓ (結果：3回目) ※1、2回目と同じ

```
rand()= 41 18467 6334 26500 19169 15724 11478 29358 26962 24464
```

▼ シードの活用②：乱数を再現する

任意のシード値を与えることで、乱数を同じ結果に再現させることができる。

通信プレイやリプレイで重宝するほか、プレイ記録を再現するようなデバッグにも役立つ。

例：シードを固定値に書き換え

```
for (int loop = 1; loop <= 3; ++loop)
{
    //シードを固定値に書き換え
    srand(123);
    printf("(loop: %d)¥n", loop);
    printf("rand()=");
    for (int i = 0; i < 10; ++i) printf(" %d", rand());
    printf("¥n");
}
```

↓ (結果)

```
(loop: 1)
rand()= 440 19053 23075 13104 32363 3265 30749 32678 9760 28064
(loop: 2) ※ 1 回目と同じ
rand()= 440 19053 23075 13104 32363 3265 30749 32678 9760 28064
(loop: 3) ※ 1、2 回目と同じ
rand()= 440 19053 23075 13104 32363 3265 30749 32678 9760 28064
```

● シード書き換えの注意点

なお、シードの書き換えは**危険がつきまとうことに注意**。

乱数の周期がリセットされるので、適正な分布が得られなくなる可能性がある。

どの処理でシードの書き換えを行うかは、慎重に判断しなければならない。チーム開発において、個人の判断でシードの書き換えを行うようなことをしてはならない。

▼ もっと乱数のランダム性を強化

もっと乱数に予測不能なばらつきを与えたい場合、毎フレーム無意味に乱数を発行するのも良い。

これをやると、再現性が不確実になるので、適用には十分注意する。

▼ 乱数生成器を自作

複数の乱数生成器の利用や自由なシードの書き換えを行うために、乱数生成器を自作するのも良い。

以下、線形合同法による乱数生成器のサンプルを示す。

例：乱数生成器のサンプル（実際の C 言語のライブラリとほぼ同じ）

```
//カスタム乱数生成器
namespace custom
{
    //線形合同法 計算式（漸化式）:  $X = (A * X(\text{前回の値}) + B) \% M$ 
    static const int SEED_INIT = 1; //初期シード
    static const int A = 1103515245; //←この A と B の値が良いと
    static const int B = 12345; // 均等分布性と周期性が良くなる
    // static const int M = 0x800000000; //32 ビット全域が対象
    static int X = SEED_INIT; //前回の値（シード）
    static const int MAX = 0x7fff; //乱数の最大値

    //乱数を取得
    int rand()
    {
        X = (A * X + B); //「% M」は省略（32 ビット全域を対象にするため）
        return (X >> 16) & RAND_MAX; //線形合同法は、下位ビットのバラつきが悪いので、上位 16 ビットを返す
    }

    //シードを更新
    void srand(const int seed)
    {
        X = seed;
        //rand(); //この時一度乱数を発行するライブラリもある
    }
}
```

▼ C++11 の乱数

C++11 では、複数の乱数生成器を用いる事や、メルセンヌ・ツイスター乱数生成器を利用することが可能。以下、そのサンプルを示す。

例：C++11 での乱数生成

```
#include <cstdint> //std::uint32_t などの型が定義
#include <random> //必要なインクルードファイル

//-----
//【乱数生成方法】

//乱数生成器のオブジェクトを作成
std::mt19937 engine;

//乱数生成器の特性を確認
std::uint32_t min = engine.min(); //乱数の最小値取得
std::uint32_t max = engine.max(); //乱数の最大値取得

//乱数生成
std::uint32_t r = engine();

//乱数生成の空回し
engine.seed(); //デフォルトシードで再初期化
engine.discard(5); //5 回乱数を生成して内部状態を進める（シードの初期化と組み合わせで一定の乱数取得）

//-----
//【分布指定による乱数生成】

//一様分布：整数
std::uniform_int_distribution<int> dist(10, 20); //10~20 の整数の一様分布（均等分布）
```

```

int r = dist(engine); //10~20 の乱数を生成

//一様分布：実数
std::uniform_real_distribution<float> dist(0.5f, 1.2f); //0.5~1.2 の実数の一様分布（均等分布）
float r = dist(engine); //0.5~1.2 の乱数を生成

//正規分布：実数（double 型）
std::normal_distribution<> dist(100.0, 20.0); //平均 100, 標準偏差 20 の実数の正規分布
double r = dist(engine); //100 付近に集中した乱数を生成、80, 120 付近を平均に 60~140 の分布が多く、その範囲外は希

//…その他いろいろな分布…

//-----
// 【乱数生成器】

//メルセンヌ・ツイスター乱数生成器（32bit 版）
//※所要バッファが大きい（約 5KB）
//※速い
std::mt19937 engine;
std::uint32_t r = engine(); //乱数生成

//メルセンヌ・ツイスター乱数生成器（64bit 版）
//※所要バッファが大きい（約 5KB）
//※速い
std::mt19937_64 engine;
std::uint64_t r = engine(); //乱数生成

//線形合同法乱数生成器（2 種類）
//※所要バッファが小さい（4 バイト）
//※速い
std::minstd_rand0 engine;
std::minstd_rand engine;
std::uint32_t r = engine(); //乱数生成

//非決定的な乱数生成器（予測不能な乱数生成器）
//※ハードウェアのノイズやマウスの動きといったものから乱数を生成する手法
//※シードの固定はできない
//※遅い
std::random_device engine;
unsigned int r = engine(); //乱数生成

//…その他いろいろな乱数生成器…

//-----
// 【シードの初期化】

//デフォルトシードで再初期化
std::mt19937 engine;
engine.seed();
std::uint32_t r = engine(); //乱数生成

//非決定乱数生成器でランダムシードを設定
//※非決定乱数生成器で毎実行時のランダム性を獲得した上で、
// 高速なメルセンヌ・ツイスターの乱数生成を使用する
std::random_device seed_gen; //非決定性乱数生成器（シード作成用）
std::mt19937 engine(seed_gen()); //エンジン生成と共にシードを与える
std::uint32_t r = engine(); //乱数生成

//シード列でより乱雑な乱数を生成
std::random_device seed_gen; //非決定性乱数生成器（シード作成用）
std::array<std::seed_seq::result_type, std::mt19937::state_size> seed_data; //シード配列
std::generate(seed_data.begin(), seed_data.end(), std::ref(seed_gen)); //シード配列を作成（乱数を与える）
//for (auto& seed : seed_data) { seed = seed_gen(); } //※このように書いてもいい
std::seed_seq seed_seq(seed_data.begin(), seed_data.end()); //シード列を生成
std::mt19937 engine(seed_seq); //シード列でエンジンを初期化

```

```
std::uint32_t r = engine(); //乱数生成
engine.seed(seed_seq);      //シードを設定し直すことも可
r = engine(); //乱数生成 (シードを与え直したので、また同じ乱数列になる)
```

■■以上■■

■ 索引

C

C++11 8

せ

線形合同法 3

め

メルセンヌ・ツイスター 3

乱数制御

以 上