Boost.Randomで
乱数を学ぼう

高橋 晶(Akira Takahashi)
id:faith_and_brave / @cpp_akira
Boost.勉強会 #13 2013/10/19(土)

はじめに

- 最近、C++標準ライブラリのリファレンスサイトcpprefjpで、<random>ヘッダのリファレンスを書いていました(完成しました！)。

  この発表では、普段mt19937とuniform_int_distributionくらいを使っている人向けに、Boost 1.54.0時点でのBoost.Random、およびC++11乱数ライブラリの全体的な解説を行っていきます。

Boost.RandomとC++11

- C++11の乱数ライブラリは、Boost.Randomを元にして作られた。
- その際、BoostからC++11にはいくつかの非互換の変更が入った。
- Boost 1.47.0でC++11の変更がBoost.Randomに適用された。
- つまり、Boost 1.47.0以降のBoost.Randomは、C++11の設計とほぼ同じ。異なる点は追々解説していくが、基本的にC++11にもある機能で解説を行っていく。

Boost C++ Libraries

乱数とは

- ランダムな数値のこと。
- ソフトウェアでは、多くの場面で乱数が必要になる
  - ボードゲームをデジタルゲーム化する際のサイコロ
  - AI
  - 抽選(おみくじ、アイテムドロップ)
  - 暗号化
  - モンテカルロ・シミュレーション
- 今回紹介するBoost.Randomでは、擬似乱数、非決定的な乱数、分布法、といった機能で、これらを実現する。

Boost C++ Libraries

話すこと

- Boost.Randomの基本的な使い方
- 擬似乱数生成器
- 非決定的な乱数生成器
- 分布器
- シード列

話さないこと

- `std::rand()`
- 数学的な詳細
- エンジンアダプタ
  - `discard_block_engine`(Boost, C++11)
  - `shuffle_order_engine`(Boost, C++11)
  - `independent_bits_engine`(Boost, C++11)
  - `addive_combine_engine`(Boost)
  - `xor_combine`(Boost)

Chapter 1

基本的な使い方

基本的な使い方

- 以下は、**メルセンヌ・ツイスター擬似乱数生成器**と**一様整数分布**を使用して、[0, 3](0以上3以下、と読む)の値を等確率で分布させる処理。

```
using namespace boost::random;

random_device seed_gen;
mt19937 engine(seed_gen());

uniform_int_distribution<> dist(0, 3);

for (int i = 0; i < 1000; ++i) {
  int result = dist(engine);
  std::cout << result << std::endl;
}
```

```
2
1
3
1
1
2
3
2
1
...
```

基本的な使い方

- 名前空間は**boost::random**。

```
using namespace boost::random;

random_device seed_gen;
mt19937 engine(seed_gen());

uniform_int_distribution<> dist(0, 3);

for (int i = 0; i < 1000; ++i) {
  int result = dist(engine);
  std::cout << result << std::endl;
}
```

```
2
1
3
1
1
2
3
2
1
…
```

基本的な使い方

- ランダムなシードで擬似乱数生成器を初期化
- 同じシードからは、同じ乱数列が生成される

```
using namespace boost::random;

random_device seed_gen;
mt19937 engine(seed_gen());

uniform_int_distribution<> dist(0, 3);

for (int i = 0; i < 1000; ++i) {
  int result = dist(engine);
  std::cout << result << std::endl;
}
```

```
2
1
3
1
1
2
3
2
1
...
```

基本的な使い方

- シードには、現在時間：エポック(1970年1月1日)からの経過時間が使われることもある。

```
using namespace boost::random;

time_t now = time(nullptr);
mt19937 engine(now);

uniform_int_distribution<> dist(0, 3);

for (int i = 0; i < 1000; ++i) {
  int result = dist(engine);
  std::cout << result << std::endl;
}
```

```
2
1
3
1
1
2
3
2
1
...
```

基本的な使い方

- 擬似乱数生成器としてmt19937を使用する。
- 32ビット版のメルセンヌ・ツイスター法。

```
using namespace boost::random;

random_device seed_gen;
mt19937 engine(seed_gen());

uniform_int_distribution<> dist(0, 3);

for (int i = 0; i < 1000; ++i) {
  int result = dist(engine);
  std::cout << result << std::endl;
}
```

```
2
1
3
1
1
2
3
2
1
...
```

基本的な使い方

- 分布法として、uniform_int_distribution(整数版の一様分布)を使用する。
- 指定した範囲の整数を等確率で発生させる。
- テンプレートパラメータは、出力する整数の型。デフォルトはint。

```
using namespace boost::random;

random_device seed_gen;
mt19937 engine(seed_gen());

uniform_int_distribution<> dist(0, 3);

for (int i = 0; i < 1000; ++i) {
  int result = dist(engine);
  std::cout << result << std::endl;
}
```

```
2
1
3
1
1
2
3
2
1
...
```

基本的な使い方

- 乱数の生成には、分布クラスの関数呼び出し演算子に、乱数生成器への参照を渡す。

```
using namespace boost::random;

random_device seed_gen;
mt19937 engine(seed_gen());

uniform_int_distribution<> dist(0, 3);

for (int i = 0; i < 1000; ++i) {
  int result = dist(engine);
  std::cout << result << std::endl;
}
```

```
2
1
3
1
1
2
3
2
1
...
```

Chapter 2

擬似乱数生成器

擬似乱数生成器

- 英語ではPseudo-Random Number Generator。
  - 略してPRNGと呼ばれることもある。
- ソフトウェアでは、規則性も再現性もない、真の乱数は作れない。
- 擬似乱数生成器は、数学的な方法で乱雑な値を生成するジェネレータ。
  - (基本的に)予測可能
  - 同じシードを設定することで、乱数列を再現できる
  - 定められた周期の分だけ乱数を生成すると、繰り返し同じ乱数列が生成される

擬似乱数生成器の一覧

linear_congruential_engine	線形合同法	Boost, C++11
mersenne_twister_engine	メルセンヌ・ツイスター法	Boost, C++11
subtract_with_carry_engine	キャリー付き減算法	Boost, C++11
inversive_congruential_engine	逆数合同法	Boost
lagged_fibonacci_engine	ラグ付きフィボナッチ法	Boost
linear_feedback_shift_engine	線形帰還シフト(暗号論的)	Boost

パラメータ設定済み擬似乱数生成器

前述した擬似乱数生成器は、多くのパラメータを設定する必要があり、
非専門家にはまず扱えない。
そのため、パラメータ設定済みのエイリアスが定義されている。

名前	説明	環境	周期	概算メモリ要件
minstd_rand0 minstd_rand	最小標準MINSTD／線形合同法 0はオリジナル、0なしは改良版	Boost, C++11	2^31 -2	int32_t
rand48	48ビット乱数	Boost	2^48-1	uint64_t
ecuyer1988	2つの線形合同法ジェネレータを組み合わせたもの	Boost	2^61	int32_t * 2
knuth_b	Knuthのリオーダーアルゴリズム B。線形合同法の順番を入れ替えたもの。 (.NET FrameworkのSystem.Randomで使われている)	Boost, C++11	?	uint32_t * 257
kreutzer1986	線形合同法の順番入れ替え	Boost	?	uint32_t * 98
taus88	2つの線形帰還シフトジェネレータをXORで組み合わせたもの	Boost	2^88	uint32_t * 3

パラメータ設定済み擬似乱数生成器

前述した擬似乱数生成器は、多くのパラメータを設定する必要があり、
非専門家にはまず扱えない。
そのため、パラメータ設定済みのエイリアスが定義される。

名前	説明	環境	周期	概算メモリ要件
hellekalek1995	線形帰還シフト	Boost	2^31 -1	int32_t
mt11213b	350次元の32ビットメルセンヌ・ツイスター(mt19937は623次元)	Boost	2^11213 -1	uint32_t * 352
mt19937	32ビット版のメルセンヌ・ツイスター	Boost, C++11	2^19937 -1	uint32_t * 625
mt19937_64	64ビット版のメルセンヌ・ツイスター	Boost, C++11	2^19937 -1	uint64_t * 312
lagged_fibonacci607 lagged_fibonacci1279 lagged_fibonacci2281 ... lagged_fibonacci44497	ラグ付フィボナッチ法。 BSD系OSのrandom関数に使われている。	Boost	2^32000 … 2^2300000	double * 607 … double * 44497

パラメータ設定済み擬似乱数生成器

前述した擬似乱数生成器は、多くのパラメータを設定する必要があり、
非専門家にはまず扱えない。
そのため、パラメータ設定済みのエイリアスが定義される。

名前	説明	環境	周期	概算メモリ要件
ranlux3 ranlux4	RANLUX法。 異なるシード間のジェネレータは乱数列が異なる、という特徴を持つ。 贅沢さレベルが0〜4あり、高くなるほど乱数の質がよくなる。 3が標準。 モンテカルロ・シミュレーションによく使われる。	Boost	10^171	int * 24
ranlux64_3 ranlux64_4	64ビット版のRANLUX法。	Boost	10^171	int64_t * 24
ranlux24 ranlux48	標準にもあるRANLUX法のtypedef。24はレベル3、48はレベル4。	Boost, C++11	10^171	uint32_t * 24 uint64_t * 12

どの擬似乱数生成器を使えばいいのか

- サイズが問題にならないのであれば、mt19937を使えばOK。
- サイズが問題になりそうなら、knuth_b、minstd_randの順で検討する。
- より高品質な乱数がほしければ、ranlux24か48。

- C++11では、非専門家がデフォルトで使用すべきtypedefとして、std::default_random_engineが定義されている(Boostにはない)。

乱数生成器コンセプト

- BoostとC++11の全ての乱数生成器は、以下のインタフェースを持っている。

```
class URNG /* Uniform Random Number Generator */ {
public:
  typedef … result_type;

  result_type operator()();

  static result_type min();
  static result_type max();
};
```

乱数生成器コンセプト

- BoostとC++11の全ての乱数生成器は、以下のインタフェースを持っている。

`result_type`	乱数生成結果の型
`operator()`	乱数を生成し、内部状態を進める。
`min()` `max()`	乱数生成する範囲。 `min()`は最小値。`max()`は最大値。 `min() <= x <= max()`の値を生成する。

擬似乱数生成器コンセプト

- BoostとC++11の全ての擬似乱数生成器は、乱数生成器コンセプトに加えて、以下のインタフェースを持っている。

```
class PRNG {
public:
  PRNG();
  PRNG(...);

  void seed(...);
  void discard(unsigend long long z); // C++11
};

bool operator==(const PRNG& a, const PRNG& b);
bool operator!=(const PRNG& a, const PRNG& b);

template <class CharT> std::basic_ostream<CharT>&
  operator<<(std::basic_ostream& os, const PRNG& x);

template <class CharT> std::basic_istream<CharT>&
  operator>>(std::basic_istream& os, PRNG& x);
```

乱数生成器コンセプト

- BoostとC++11の全ての乱数生成器は、以下のインタフェースを持っている。

`PRNG()`	デフォルトのシード(固定値)で内部状態を初期化する。 自動的に時間や乱数で初期化するわけではない。
`PRNG(...)`	パラメータを受け取って内部状態を初期化する。 C++11では、シードシーケンス、シード値を受け取ることが規定されている。
`seed(...)`	シードを指定し、内部状態を再初期化する。 コンストラクタと同じパラメータを受け取る。
`discard(z)`	内部状態をz回進める。 別な言い方をすれば、z回だけoperator()の結果を捨てる。C++11でのみ規定される。
`a == b`	等値比較。 aとbが持つ状態シーケンスが全て同じならtrueを返す。
`a != b`	非等値比較。 aとbが持つ状態シーケンスが異なればtrueを返す。

乱数生成器コンセプト

- BoostとC++11の全ての乱数生成器は、以下のインタフェースを持っている。

`os << x`	シリアライズ。 出力ストリームに、現在状態を出力する。 フォーマットは未規定(なので、C++11の方は異なる環境間でデシリアライズはできない)。
`is >> x`	デシリアライズ。 入力ストリームから、現在状態を読み込む。

擬似乱数生成器の状態を保存・復旧する

- ひとつめの方法は、ストリーム演算子によるシリアライズ。
  - これは非常に簡単。
  - しかし、擬似乱数生成器の種類のよっては、巨大な状態シーケンス(mt19937だと623要素の配列 + 1)を出力するので、保存データがそこそこ大きくなる可能性がある。
  - また、C++11では出力フォーマットの環境差異がありえる。

```
boost::random::mt19937 engine;
std::cout << engine << std::endl;
```

Boost C++ Libraries

擬似乱数生成器の状態を保存・復旧する

621461756 1301868182 2938499221 2950281878 1875628136 751856242 944701696 2243192071 694061057 219885934 2066767472 3182869408 485472502 2336857883
1071588843 3418470598 951210697 3693558366 2923482051 1793174584 2982310801 1586906132 1951078751 1808158765 1733897588 431328322 4202539044 530658942
1714810322 3025256284 3342585396 1937033938 2640572511 1654299090 3692403553 4233871309 3497650794 862629010 2943236032 2426458545 1603307207
1133453895 3099196360 2208657629 2747653927 931059398 761573964 3157853227 785880413 730313442 124945756 2937117055 3295982469 1724353043 3021675344
3884886417 4010150098 4056961966 699635835 2681338818 1339167484 720757518 2800161476 2376097373 1532957371 3902664099 1238982754 3725394514 3449176889
3570962471 4287636090 4087307012 3603343627 202242161 2995682783 1620962684 3704723357 371613603 2814834333 2111005706 624778151 2094172212 4284947003
1211977835 991917094 1570449747 2962370480 1259410321 170182696 146300961 2836829791 619452428 2723670296 1881399711 1161269684 1675188680 4132175277
780088327 3409462821 1036518241 1834958505 3048448173 161811569 618488316 44795092 3918322701 1924681712 3239478144 383254043 4042306580 2146983041
3992780527 3518029708 3545545436 3901231469 1896136409 2028528556 2339662006 501326714 2060962201 2502746480 561575027 581893337 3393774360 1778912547
3626131687 2175155826 319853231 986875531 819755096 2915734330 2688355739 3482074849 2736559 2296975761 1029741190 2876812646 690154749 579200347
4027461746 1285330465 2701024045 4117700889 759495121 3332270341 2313004527 2277067795 4131855432 2722057515 1264804546 3848622725 2211267957
4100593547 959123777 2130745407 3194437393 486673947 1377371204 17472727 352317554 3955548058 159652094 1232063192 3835177280 49423123 3083993636 733092
2120519771 2573409834 1112952433 3239502554 761045320 1087580692 2540165110 641058802 1792435497 2261799288 1579184083 627146892 2165744623 2200142389
2167590760 2381418376 1793358889 3081659520 1663384067 2009658756 2689600308 739136266 2304581039 3529067263 591360555 525209271 3131882996 294230224
2076220115 3113580446 1245621585 1386885462 3203270426 123512128 12350217 354956375 4282398238 3356876605 3888857667 157639694 2616064085 1563068963
2762125883 4045394511 4180452559 3294769488 1684529556 1002945951 3181438866 22506664 691783457 2685221343 171579916 3878728600 2475806724 2030324028
3331164912 1708711359 1970023127 2859691344 2588476477 2748146879 136111222 2967685492 909517429 2835297809 3206906216 3186870716 341264097 2542035121
3353277068 548223577 3170936588 1678403446 297435620 2337555430 466603495 1132321815 1208589219 696392160 894244439 2562678859 470224582 3306867480
201364898 2075966438 1767227936 2929737987 3674877796 2654196643 3692734598 3528895099 2796780123 3048728353 842329300 191554730 2922459673 3489020079
3979110629 1022523848 2202932467 3583655201 3565113719 587085778 4176046313 3013713762 950944241 396426791 3784844662 3477431613 3594592395 2782043838
3392093507 3106564952 2829419931 1358665591 2206918825 3170783123 31522386 2988194168 1782249537 1105080928 843500134 1225290080 1521001832 3605886097
2802786495 2728923319 3996284304 903417639 1171249804 1020374987 2824535874 423621996 1988534473 2493544470 1008604435 1756003503 1488867287 1386808992
732088248 1780630732 2482101014 976561178 1543448953 2602866064 2021139923 1952599828 2360242564 2117959962 2753061860 2388623612 4138193781 2962920654
2284970429 766920861 3457264692 2879611383 815055854 2332929068 1254853997 3740375268 3799380844 4091048725 2006331129 1982546212 686850534 1907447564
2682801776 2780821066 998290361 1342433871 4195430425 607905174 3902331779 2454067926 1708133115 1170874362 2008609376 3260320415 2211196135 433538229
2728786374 2189520818 262554063 1182318347 3710237267 1221022450 715966018 2417068910 2591870721 2870691989 3418190842 4238214053 1540704231 1575580968
2095917976 4078310857 2313532447 2110690783 4056346629 4061784526 1123218514 551538993 597148360 4120175196 3581618160 3181170517 422862282 3227524138
1713114790 662317149 1230418732 928171837 1324564878 1928816105 1786535431 2878099422 3290185549 539474248 1657512683 552370646 1671741683 3655312128
1552739510 2605208763 1441755014 181878989 3124053868 1447103986 3183906156 1728556020 3502241336 3055466967 1013272474 818402132 1715099063 2900113506
397254517 4194863039 1009068739 232864647 2540223708 2608288560 2415367765 478404847 3455100648 3182600021 2115988978 434269567 4117179324 3461774077
887256537 3545801025 286388911 3451742129 1981164769 786667016 3310123729 3097811076 2224235657 2959658883 3370969234 2514770915 3345656436 2677010851
2206236470 271648054 2342188545 4292848611 3646533909 3754009956 3803931226 4160647125 1477814055 4043852216 1876372354 3133294443 3871104810
3177020907 2074304428 3479393793 759562891 164128153 1839069216 2114162633 3989947309 3611054956 1333547922 835429831 494987340 171987910 1252001001
370809172 3508925425 2535703112 1276855041 1922855120 835673414 3030664304 613287117 171219893 3423096126 3376881639 2287770315 1658692645 1262815245
3957234326 1168096164 2968737525 2655813712 2132313144 3976047964 326516571 353088456 3679188938 3205649712 2654036126 1249024881 880166166 691800469
2229503665 1673458056 4032208375 1851778863 2563757330 376742205 1794655231 340247333 1505873033 396524441 879666767 3335579166 3260764261 3335999539
506221798 421658741 975887814 2080536343 3360539560 571586418 138896374 4234352651 2737620262 3928362291 1516365296 38056726 3599462320 3585007266
3850961033 471667319 1536883193 2310166751 1861637689 2530999841 4139843801 2710569485 827578615 2012334720 2907369459 3029312804 2820112398 1965028045
35518606 2478379033 643747771 1924139484 4123405127 3811735531 3429660832 3285177704 1948416081 1311525291 1183517742 1739192232 3979815115 2567840007
4116821529 213304419 4125718577 1473064925 2442436592 1893310111 4195361916 3747569474 828465101 2991227658 750582866 1205170309 1409813056 678418130
1171531016 3821236156 354504587 4202874632 3882511497 1893248677 1903078632 26340130 2069166240 3657122492 3725758099 831344905 811453383 3447711422
2434543565 4166886888 3358210805 4142984013 2988152326 3527824853 982082992 2809155763 190157081 3340214818 2365432395 2548636180 2894533366 3474657421
2372634704 2845748389 43024175 2774226648 1987702864 3186502468 453610222 4204736567 1392892630 2471323686 2470534280 3541393095 4269885866 3909911300
759132955 1482612480 667715263 1795580598 2337923983 3390586366 581426223 1515718634 476374295 705213300 363062054 2084697697 2407503428 2292957699
2426213835 2199989172 1987356470 4026755612 2147252133 270400031 1367820199 2369854699 2844269403 79981964

擬似乱数生成器の状態を保存・復旧する

- ふたつめの方法では、シードと生成回数を覚えておいて、復旧させる。
- シードは擬似乱数生成器自身は管理しないので、指定するユーザーが管理する。
- 分布クラス内で、擬似乱数生成器のoperator()が複数回呼ばれるので、何回生成されたか覚えておくための、擬似乱数生成器ラッパーを用意しておくとよい。
- この方法の問題点は、後述する分布クラスもまた状態を持っているということ。分布の状態は、この方法では復旧できない。ストリームなら可能。

```
// 保存しておいたシードと生成回数
unsigned long long seed = 0x5eed;
unsigned long long count = 5;

boost::random::mt19937 engine(seed);
engine.discard(count);
```

Chapter 3

非決定的な乱数生成器

非決定的な乱数生成器

- 英語ではNon-deterministic Random Number Generator。
- ソフトウェアでは真の乱数を作れないので、ハードウェアに乱数を作らせるというもの。以下のようなものをエントロピー(乱雑さ)の素材として使用する：
  - ノイズ、熱雑音、放射線センサ、マウスの動き
- 非決定的な乱数は、暗号化のようなセキュリティ用途に使用する。
- BoostとC++11では、`random_device`という型で、非決定的な乱数を提供する。
- Windows では`CryptGenRandom()` API、UNIX系OSでは/dev/urandomや/dev/randomといった特殊なファイルを使用して実装される。
- ハードウェアからのエントロピー取得やファイルI/Oでそんなに早くないので、必要ないところでむやみに使用しないこと。

パスワード生成

- 非決定的な乱数を使用した、パスワード生成

```
using namespace boost::random;

// パスワードに使用する文字
std::string chars(
    "abcdefghijklmnopqrstuvwxyz"
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    "1234567890"
    "!@#$%^&*()"
    "`~-_=+[{]{\\|;:'\",<.>/? ");

random_device rng;

// ランダムに8文字選択する
uniform_int_distribution<> index_dist(0, chars.size() - 1);
for (int i = 0; i < 8; ++i) {
    std::cout << chars[index_dist(rng)];
}
std::cout << std::endl;
```

random_deviceのインタフェース

- random_deviceクラスは、乱数生成器コンセプトに加えて、以下のインタフェースを持つ。

`explicit random_device(const std::string& token);`	何らかの文字列を受け取るコンストラクタ。 動作は実装定義だが、”/dev/random”を明示的に指定して乱数の取得元を切り替えたりするのに使用する。 BoostでのUNIX系OSのデフォルトは”/dev/urandom”
`double entropy() const;`	エントロピー(乱雑さ)を取得する。乱雑さがない(擬似乱数で実装される)場合は0を返す。 Boostは固定値で10を返す。 libstdc++とlibc++はなぜか0を返す。
`template <class Iterator>` `void generate(Iterator first, Iterator last);`	範囲に対して乱数を生成する。

Chapter 4

乱数分布

乱数分布

- 英語ではRandom Number Distribution。
- 通常、乱数生成器が生成する乱数を、そのまま使うことはしない。
- 各分布法によって、各分野、各目的で必要となる乱数を生成する。
- ほとんどの分布法には、現実世界に対応する事象が存在する。
  - そうでないのは、より一般化したものや、値の性質分析に使用する。
- ここでは、分布の一覧を見たあと、いくつかの実際の使用例を紹介する。

Boost C++ Libraries

乱数分布の一覧

`uniform_int_distribution<UIntType = int>`	一様分布の整数版。 [min, max]の範囲を等確率で発生させる。 用途例：サイコロ	Boost, C++11
`uniform_real_distribution<RealType = double>`	一様分布の実数版。 [min, max)の範囲を等確率で発生させる。 用途例：[0, 2π)の範囲で、てきとうに角度を決定する	Boost, C++11

Boost C++ Libraries

乱数分布の一覧

`bernoulli_distribution`	ベルヌーイ分布。 trueとfalseの2値を、確率指定して発生させる。 用途例：コイントス	Boost, C++11
`binomial_distribution<IntType = int>`	二項分布。 成功する確率pの事象をn回施行し、成功した回数を求める。 用途例：N個の製品に、確率pで不良品が発生する統計をとる	Boost, C++11
`geometric_distribution<IntType = int>`	幾何分布。 成功する確率pの事象が、初めて成功するまでに何回失敗したかを求める。	Boost, C++11
`negative_binomial_distribution<IntType = int>`	負の二項分布。 成功する確率pの事象がk回成功するまでに失敗した回数を求める。	Boost, C++11

Boost C++ Libraries

乱数分布の一覧

`poisson_distribution<IntType = int>`	ポワソン分布。 用途例：この資料を書いている間に変換間違えをする数の統計をとる。	Boost, C++11
`exponential_distribution<RealType = double>`	指数分布。 用途例：銀行の窓口に顧客が到着する時間間隔の統計をとる。	Boost, C++11
`weibull_distribution<RealType = double>`	ワイブル分布。 用途例：物体の劣化、寿命、強度、破壊の統計をとる。	Boost, C++11
`extreme_value_distribution<RealType = double>`	極値分布。 用途例：来年の最高気温予測。	Boost, C++11

Boost C++ Libraries

乱数分布の一覧

`normal_distribution<RealType = double>`	正規分布。 用途例：平均身長170cm、標準偏差5cmで、平均付近の値を生成させる。	Boost, C++11
`lognormal_distribution<RealType = double>`	対数正規分布。 用途例：社員の給与アップの統計(スタートは同じでも、給与が上がる人数は減少していく)。その他、株価変動など。	Boost, C++11
`chi_squared_distribution<RealType = double>`	カイ二乗分布。 用途例：赤玉20個、白玉20個が混ざった箱から、20個の玉を取り出す。その際の、期待値(赤玉10個、白玉10個)からのずれの度合いを判定する。	Boost, C++11

Boost C++ Libraries

乱数分布の一覧

`cauchy_distribution<RealType = double>`	コーシー分布。 用途例：正規分布に比べて外れ値(outliers)が非常に多い場合の分布のモデルに、使われることがある。	Boost, C++11
`fisher_f_distribution<RealType = double>`	F分布。 用途例：2つの標本の分散比を求める	Boost, C++11
`student_t_distribution<RealType = double>`	t分布。 用途例：母集団の性質(平均、分散)を推定する	Boost, C++11

Boost C++ Libraries

乱数分布の一覧

`discrete_distribution<IntType = int>`	確率列を指定し、選択されたインデックスを得る。 用途例：おみくじ。ガチャ。AIの行動選択。	Boost, C++11
`piecewise_constant_distribution<` `  RealType = double>`	区間ごとに重み付けする。	Boost, C++11
`piecewise_linear_distribution<` `  RealType = double>`	区間ごとの重み付けを線形に変化させる。	Boost, C++11

乱数分布の一覧

`triangle_distribution<RealType = double>`	三角分布。 用途例：納期の期待値。	Boost

`uniform_on_sphere<`
`  RealType = double,`
`  Cont = std::vector<RealType>>`
	球状の一様分布。
	用途例：地球上のランダムな位置を選択する。
		Boost

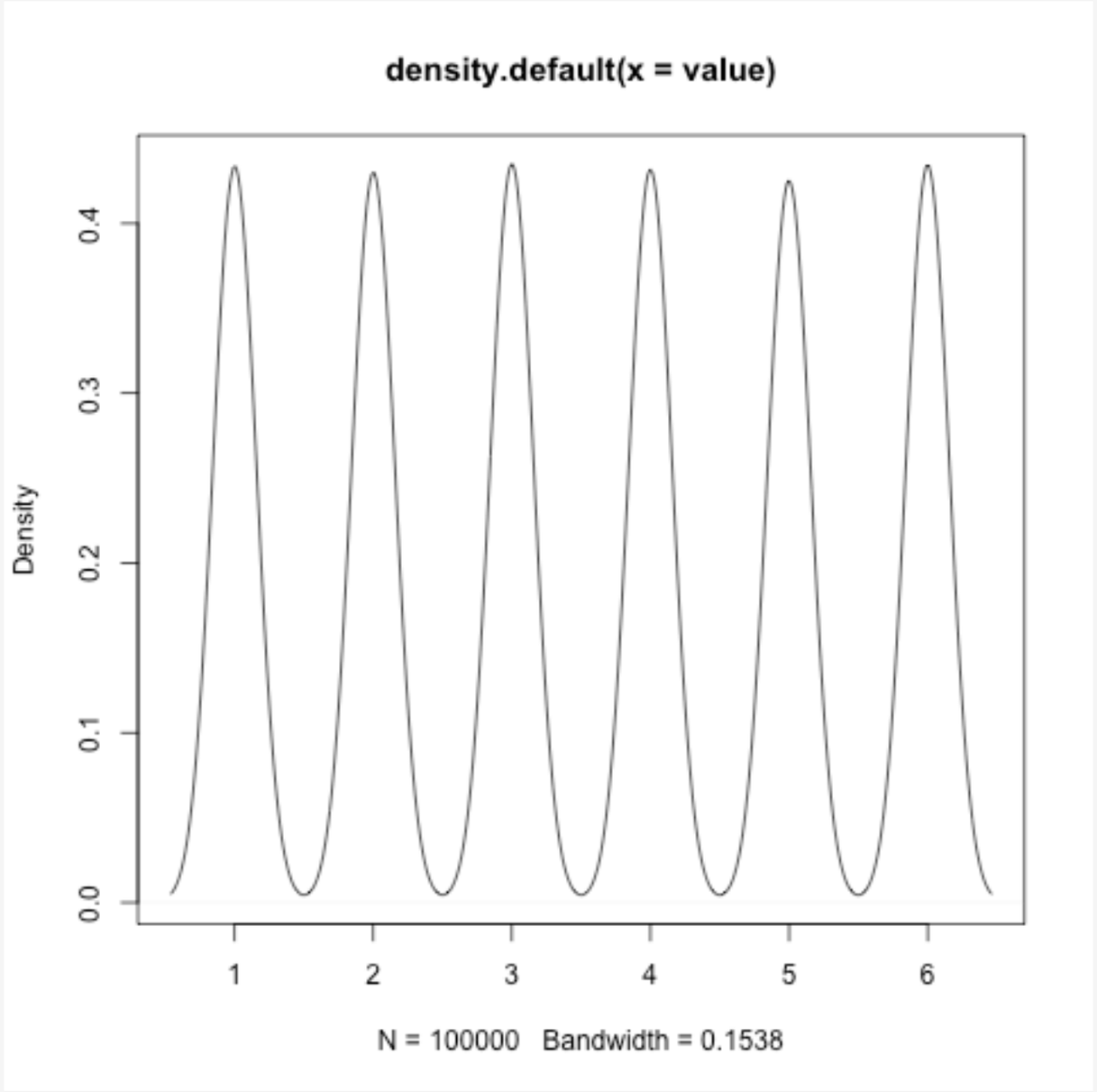
一様整数分布の例：サイコロ

- 範囲[1, 6]の整数を等確率で分布させ、サイコロを作る。

```
using namespace boost::random;

random_device seed_gen;
mt19937 engine(seed_gen());
uniform_int_distribution<std::size_t> dist(1, 6);

std::size_t selected = dist(engine);
std::cout << selected << std::endl;
```

一様整数分布の例：サイコロ

- 統計をとってみる。値が偏りなく分布していることがわかる。



正規分布の例：人物データの自動生成

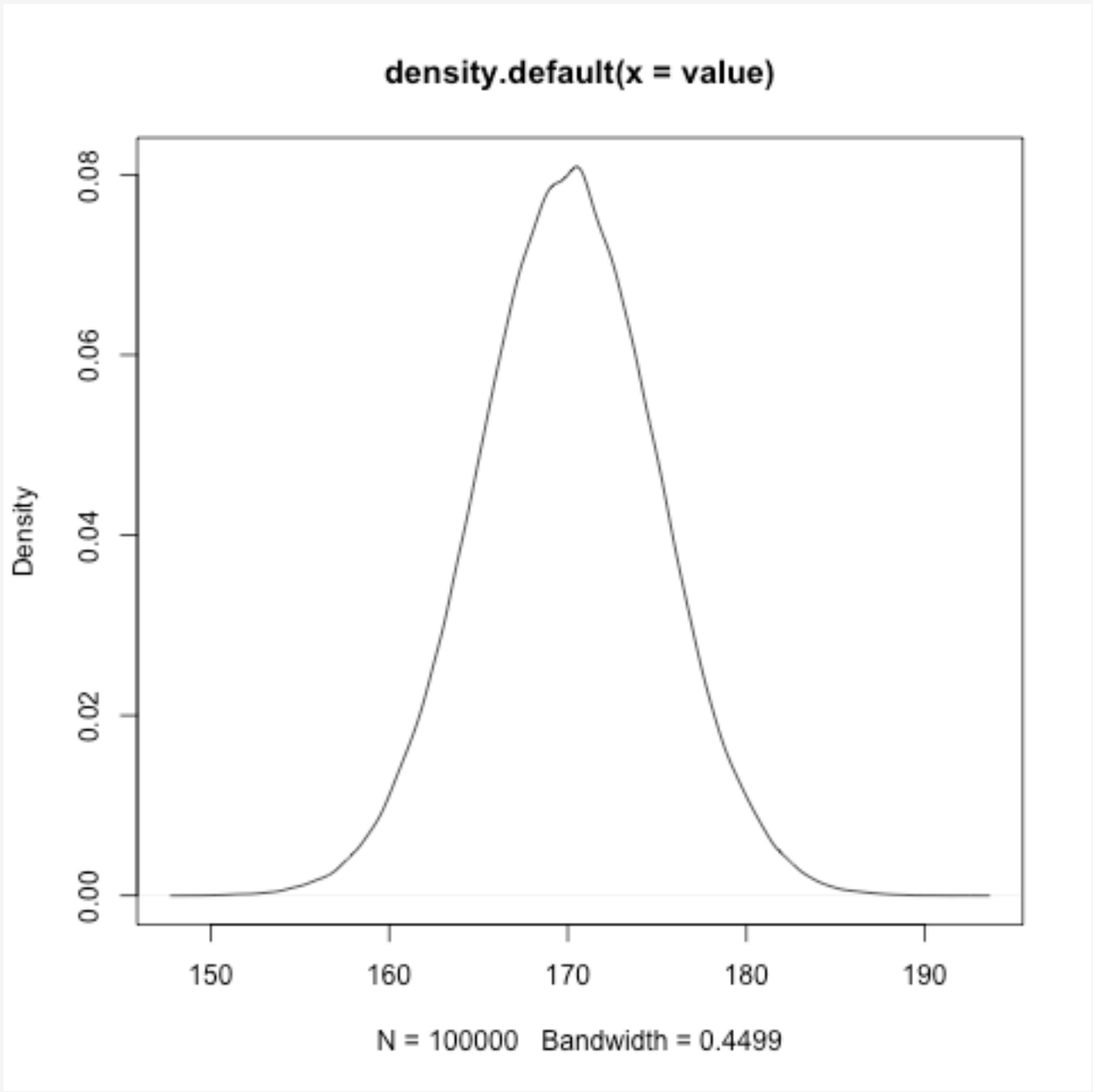
- 平均身長170cm、標準偏差5cmで、平均付近の身長データを自動生成する

```
using namespace boost::random;

random_device seed_gen;
mt19937 engine(seed_gen());

normal_distribution<> dist(170.0, 5.0);

for (int i = 0; i < 10; ++i) {
  double hight = dist(engine);
  std::cout << hight << std::endl;
}
```

```
165.746
177.708
164.219
170.475
168.85
174.766
175.37
159.845
178.156
174.682
```

正規分布の例：人物データの自動生成

- 統計をとってみる。平均身長170cmを中心に、山なりになっている。
  - たまに外れ値(平均 ± 標準偏差より離れている値)も生成される。



discrete_distributionの例：おみくじ

- 確率列に基いて、0から始まるインデックスを生成する。おみくじ確率の参考：「おみくじで大吉を当てる確率」
  http://r25.yahoo.co.jp/fushigi/wxr_detail/?id=20111231-00022429-r25

```
random_device seed_gen;
mt19937 engine(seed_gen());

const std::vector<std::pair<std::string, double>> table = {
  {"大吉",   17},
  {"吉",     35},
  {"半吉",    5},
  {"小吉",    4},
  {"末小吉",  3},
  {"末吉",    6},
  {"凶",     30}
};

discrete_distribution<std::size_t> dist(table | map_values);
for (int i = 0; i < 10; ++i) {
  std::size_t index = dist(engine);
  std::cout << table[index].first << std::endl;
}
```

```
大吉
吉
小吉
凶
半吉
大吉
吉
吉
吉
凶
```

ちょっとした実験

- 分布クラスがてきとうに値をばらけさせてくれるなら、乱数生成器はそれほど性能がよくなくてもいいんじゃないか？テキトウな値するエセ乱数生成器を作ってやってみよう。

```
class sequence_engine {
  unsigned int state_;
public:
  typedef unsigned int result_type;

  sequence_engine(result_type seed = 0)
    : state_(seed) {}

  result_type operator()()
    { return state_ += 123459678; }

  result_type min() const
    { return 0; }

  result_type max() const
    { return std::numeric_limits<result_type>::max(); }
}
```

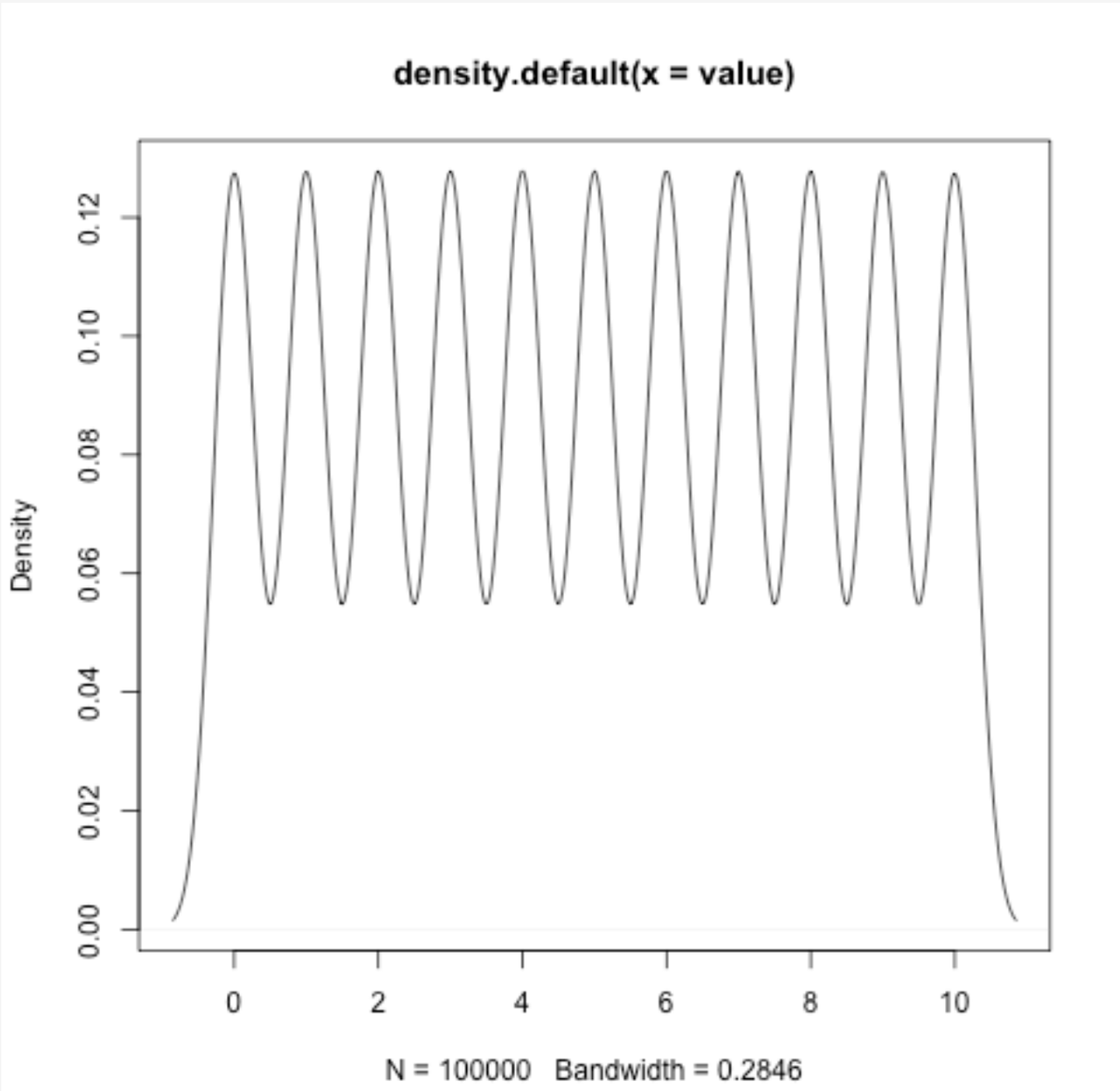
ちょっとした実験

- 分布クラスがてきとうに値をばらけさせてくれるなら、乱数生成器はそれほど性能がよくなくてもいいんじゃないか？テキトウな値を生成するエセ乱数生成器を作ってやってみよう。

```
sequence_engine engine;
boost::random::uniform_int_distribution<> dist(0, 10);

for (int i = 0; i < 100000; ++i) {
  std::cout << dist(engine) << std::endl;
}
```

ちょっとした実験：統計をとってみる

- お、等確率で分布してるぞ！



ちょっとした実験：統計をとってみる

- しかし、値の順番は残念でした。分布には、良質な乱数生成器が必要です。

```
0
0
0
1
1
1
2
2
2
3
3
3
4
4
4
5
5
5
6
6
6
6
```

分布コンセプト

- 全ての分布クラスは、以下のインタフェースを持っている。

```
class Distribution {
public:
  typedef … result_type;
  typedef … param_type; // C++11

  Distribution();                          // C++11
  Distribution(const param_type& parm); // C++11

  void reset();

  template <class Engine>
  result_type operator()(Engine& e);
  template <class Engine>
  result_type operator()(Engine&, e, const param_type& parm); // C++11

  param_type param() const;            // C++11
  void param(const param_type& parm); // C++11

  result_type min() const; // C++11
  result_type max() const; // C++11
};

template <class CharT>
std::basic_ostream<CharT> operator<<(std::basic_ostream<CharT>& os, const Distribution&);

template <class CharT>
std::basic_istream<CharT> operator>>(std::basic_istream<CharT>& is, Distribution&);
```

分布コンセプト

- 全ての分布生成器は、以下のインタフェースを持っている。
- C++11のものは、Boostにドキュメントとして明記されていないだけで、Boostも共通インタフェースとして持っている。

`result_type`	乱数生成結果の型
`param_type`	分布パラメータをまとめた型(C++11)
`Distribution()`	デフォルトコンストラクタ(C++11)
`Distribution(parm)`	パラメータをまとめて受け取るコンストラクタ(C++11)
`reset()`	内部状態をリセットする
`operator(e)`	乱数生成器コンセプトを満たすエンジンへの参照を受け取り、分布された値を生成する。
`operator(e, parm)`	コンストラクタで設定されたパラメータではなく、乱数生成時にパラメータを渡す(C++11)。 ※状態を持たないというわけではない

分布コンセプト

- 全ての分布生成器は、以下のインタフェースを持っている。
- C++11のものは、Boostにドキュメントとして明記されていないだけで、Boostも共通インタフェースとして持っている。

min() max()	発生しうる値の範囲(C++11)。 min()は最小値。max()は最大値。 min() <= x <= max()の値を生成する。 パラメータによって範囲が変わるため、非静的メンバ関数となっている。
param()	パラメータオブジェクトを取得する(C++11)
param(parm)	パラメータを再設定する(C++11)
os << x	シリアライズ。 出力ストリームに、現在状態を出力する。 フォーマットは未規定(なので、C++11の方は異なる環境間でデシリアライズはできない)。
is >> x	デシリアライズ。 入力ストリームから、現在状態を読み込む。

パラメータオブジェクト

- 分布クラスは、それぞれ異なるパラメータを持っている。
- 共通インタフェース化を可能にするため、分布クラスのコンストラクタと同じパラメータを持つ、param_typeを全ての分布クラスに持たせている。

```
using namespace boost::random;

random_device seed_gen;
mt19937 engine(seed_gen());

using dist_type = uniform_int_distribution<>;

// パラメータオブジェクトに、分布パラメータを設定する
dist_type::param_type param(1, 6);

// パラメータをまとめて分布オブジェクトに設定する
dist_type dist(param);

for (int i = 0; i < 10; ++i) {
  std::cout << dist(engine) << std::endl;
}
```

Chapter 5

シード列

シード列

- より多くの乱雑さが必要な場合のために、単一のシード値ではなく、複数のシード値を与えるseed_seqクラスが、BoostとC++11で提供されている。
- このクラスは内部的に、32ビット整数としてのシードのstd::vectorである(と決まっている)。

seed_seqの使い方例

- 32ビット版メルセンヌ・ツイスターの623次元 + 1の状態シーケンスを、その次元分のシード列で初期化する。

```
using namespace boost::random;

// 擬似乱数生成器の状態シーケンスのサイズ分、
// シードを用意する
boost::array<
  seed_seq::result_type,
  mt19937::state_size
> seed_data;

// 非決定的な乱数でシード列を構築する
random_device seed_gen;
boost::generate(seed_data, std::ref(seed_gen));

seed_seq seq(seed_data.begin(), seed_data.end());

// 擬似乱数生成器をシード列で初期化
mt19937 engine(seq);

// 乱数生成
for (int i = 0; i < 10; ++i) {
  std::uint32_t result = engine();
  std::cout << result << std::endl;
}
```

おわりに

- 今回の発表では、Boost.Randomの全体的な紹介をしました。
- 一口に乱数といっても、いろいろな用途があり、その用途に特化した実装がときに必要になります。
  Boost.Randomは、様々な用途を想定して設計され、用途別の機能が提供されています。
- 何かひとつでも、新しくわかったことがあれば幸いです。

参照

- Boost Random Library
  http://www.boost.org/doc/libs/release/doc/html/boost_random.html
- <random> - cpprefjp - C++ Reference Site
  https://sites.google.com/site/cpprefjp/reference/random
- さまざまな確率分布 (probability distributions)
  http://www.biwako.shiga-u.ac.jp/sensei/mnaka/ut/statdist.html
- 確率変数ー種々の分布の特徴
  https://sites.google.com/site/techdmba/distribution
- N1398 A Proposal to Add an Extensible Random Number Facility to the Standard Library
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1398.html