Abstract

This paper highlights Pseudo Random Number Generators (PRNGs) and how to use common PRNGs found in C++ libraries to generate random numbers with arbitrary precision. We look at the current PRNGs available in the C++ libraries, mt19937, ranlux24, ranlux48, and others plus we show the implementations of new ones like the Xoshiro family of PRNGs and the ChaCha20 PRNG. The latter is graded for cryptographic applications.

Introduction

Most PRNGs available delivered an unsigned integer of either 32-bit or 64-bit. When the quality of these PRNGs is high enough it can be used to expand it into arbitrary precision PRNGs. We start by looking at what is available in the standard C++ random library. We discard most of them as insufficient in today's standard and then we look at newer versions like the Xoshiro family of PRNGs and of course, Bernstein's Chacha20 whose quality is also sufficient for many cryptographic applications. Since the PRNGs in the C++ are built as classes, we will build Xoshiro's and ChaCha20 using the same class structure and that allows us to expand into the arbitrary precision realm where we create a template-based class that uses some of the underlying PRNGs to create arbitrary precision random numbers. We finally look at the performance of these methods and provide a recommendation.

Table of Contents

Contents

Abstract 1
Introduction 1
Table of Contents 2
The Arbitrary precision library 3
Int_precision class 3
Internal format for int_precision variables 3
Pseudo Random Number Generation (PRNG) Algorithm in the C++ standard library 4
The Mersenne Twister 5
Initializing the Mersenne Twister algorithm 6
The Xoshiro family of PRNG 7
Source code for Xoshiro256** class 8
The ChaCha family of PRNG 9
Initialization of the ChaCha20 PRNG involves three distinct keys: 10
Is ChaCha20 considered cryptographic-grade? 11
Source code for Chacha20 class 12
The arbitrary precision version of a template-based PRNG. 15
Performance 17
Recommendation 20
Reference 21
Appendix 22
Source Xoshiro Class 22

The Arbitrary precision library

If you already are familiar with the arbitrary precision library, you can skip this section. There are two classes. One for *int_precision* that handle arbitrary precision integers and one for *float_precision* that handles all *floating-point* arbitrary precision. Since Prime numbers are integers, we only need to highlight the int_precision class.

Int_precision class

To understand the C++ code and text we have to highlight a few features of the arbitrary precision library where the class name is *int_precision*. Instead of declaring, a variable with any of the build-in integer type char, short, int, long, long long, unsigned char, unsigned short, unsigned int, unsigned long, and unsigned long long you just replace the type name with *int_precision*. E.g.

```
int_precision ip;  // Declare an arbitrary precision integer
```

You can do any integer operations with *int_precision* that you can do for any type of integer in C++. Furthermore, there are a few methods you will need to know.

One of them is .iszero() which simply returns true or false if the *int_precision* variable is zero or not zero. Another is .even() and .odd() which return the Boolean value of the number even and odd status. There are other methods but I will refer you to the user manual for the arbitrary precision package [1].

Internal format for int_precision variables

For the internal layout of the arbitrary precision number, we are using the STL vector library declared as:

```
vector<uintmax_t> mNumber;
```

uintmax_t is mostly a 64-bit quantity on most systems, so we use a vector of 64-bit unsigned integer to store our integer precision number.

The method .size() returns the number of internal vector entries needed to hold the number.

The number is stored such that the vector mNumber[0] holds the least significant 64-bit binary data. The mNumber[size()-1] holds the most significant 64-bit binary data. The sign is kept separately in a class field variable mSign, which means that the mNumber holds the unsigned binary vector data.

For more details see [1].

Pseudo-Random Number Generation (PRNG) Algorithm in the C++ standard library

In [2] & [3] there is an easy introduction to the various *Pseudo-Random Number Generation* (PRNG) algorithms available in the C++ standard library. Generally speaking, many of the PRNGs are not useful in real practice except maybe for the Mersenne twister PRNG. Below is the table of the capability of the C++20 standard from [2].

Type name	Family	Period	State size in bytes	Performance	Quality	Usefulness
minstd_rand	Linear congruential generator	2^{31}	4	Bad	Awful	No
mt19937 **mt19937_64**	Mersenne twister	2^{19937}	2500	Decent	Decent	Probably
ranlux24 **ranlux48**	Subtract and carry	10^{171}	96	Awful	Good	No
knuth_b	Shuffle linear congruential generator	2^{31}	1024	Awful	Bad	No
default_random_engine	Any of the above (implementation-defined)	varies	varies	varies	varies	
rand()	Linear congruential generator	2^{31}	4	Bad	Awful	No

As recommended in [2], the preferred choice for a pseudorandom number generator (PRNG) is the Mersenne Twister. However, it is also worth considering other PRNGs that are not part of the standard C++ Library. In this section, we will discuss the implementation of the following PRNGs:

1. Mersenne Twister
2. Xoshiro family of scrambled linear PRNGs
3. ChaCha family of cryptographic-strength PRNGs

Please note that PRNGs 1-2 are only recommended for non-cryptographic purposes. While there are C++ sources available for many of these PRNGs, they are typically designed to work with either 32-bit or 64-bit versions. Therefore, when using an arbitrary precision version, certain modifications are required to ensure compatibility.

Most PRNGs in the C++ library are structured around classes, and we will adhere to this design approach when creating arbitrary precision versions. This allows us to utilize the same available methods and design philosophy as the built-in versions, making it easier to transition from a 32-bit or 64-bit environment to the arbitrary precision version. Generally, the following methods are available for most PRNGs:

1. The constructor of the class: This initializes the PRNG with either a default seed or a seed obtained from the *seed_seq* class.
2. The () operator: This generates the next pseudorandom number.

3. Member methods:
   a. seed(): Allows manual seeding of the PRNG.
   b. min(): Returns the minimum number that the PRNG can generate.
   c. max(): Returns the maximum number that the PRNG can generate.
   d. discard(): Discards a specified number of subsequent pseudorandom numbers.
4. Non-member methods:
   a. == relational operator
   b. != relational operator

Sometimes PRNGs require a warm-up period to improve the quality of the generated random numbers. This can be achieved by discarding a certain number of initial pseudo-random numbers using the discard() member function or by utilizing the seed_seq class to provide better initial seeding.

It's important to note that all non-cryptographic PRNGs mentioned here are deterministic. This means that if we seed the PRNG with the same value, it will produce the same sequence of numbers each time. For a beginner-friendly introduction to PRNGs, refer to [2] and [3].

When transitioning from the 32-bit or 64-bit versions of PRNGs to arbitrary precision, an additional parameter is needed for generating the next pseudo-random number. This parameter represents the maximum size of the PRNG to be generated. The C++ standard library sets a fixed maximum size of 64 bits for the PRNG. However, in arbitrary precision scenarios, this limit doesn't exist, so we must specify the maximum size in bits for the PRNG to generate. Consequently, the method 'max(size_t max_bits = 64)' and the generator 'operator()(size_t max_bits = 64)' need to include a parameter for the maximum size in bits. We have chosen to provide some default size in case the parameter is omitted, in which case the class will revert to the standard behavior of the C++ library.

In the following sections, we will introduce each PRNG, providing a brief history and simplified explanations, before delving into their arbitrary precision implementations.

The Mersenne Twister

The Mersenne Twister is a widely used pseudo-random number generator (PRNG) algorithm known for its long period and good statistical properties. It was developed by Makoto Matsumoto and Takuji Nishimura in 1997.

The Mersenne Twister is based on a large Mersenne prime number, which is a prime number in the form 2^p - 1, where p is also a prime number. The Mersenne Twister uses a 32-bit variant of the Mersenne Prime, specifically the Mersenne Prime with p = 19937.

There are four key features and properties of the Mersenne Twister:

1. *Period*: The Mersenne Twister has a period of 2^{19937} - 1, which means it can generate 2^{19937} - 1 distinct random number before repeating. This period is extremely large, ensuring a vast number of unique random values.

2. *Uniformity*: The generated random numbers by the Mersenne Twister are uniformly distributed over the entire range of possible values. Each value has an equal chance of being generated.
3. *Speed*: The Mersenne Twister is known for its relatively fast generation speed. Although it is not the fastest PRNG available, it strikes a good balance between speed and quality of randomness.
4. *State and Seeding*: The Mersenne Twister has an internal state of 624 32-bit integers. The state determines the current position in the sequence of random numbers. By default, when you create an instance of the Mersenne Twister engine, it is seeded with a value obtained from the system clock. However, you can also manually seed it with a specific value.

However, The Mersenne Twister has also some weaknesses like it is predictable and fails some statistical tests.

To use the Mersenne Twister PRNG in C++ with the standard library, you can instantiate the \`std::mt19937\` or \`std::mt19937_64\` classes from the \`<random>\` header, depending on whether you want a 32-bit or 64-bit variant. You can then generate random numbers by calling member functions \`operator()\`.

Initializing the Mersenne Twister algorithm

The \`std::mt19937\` class uses a 32-bit version of the Mersenne prime with p = 19937. The internal state of the Mersenne Twister consists of an array of 624 32-bit integers. These integers are collectively referred to as the "state vector."

To initialize the state vector of the Mersenne Twister, it requires a seed value. The seed value is used to generate the initial state by applying a process called the "twist" operation. The twist operation helps to ensure that the generated sequence of random numbers exhibits good statistical properties and has a long period.

The seed value passed to the \`std::mt19937\` constructor or the \`seed()\` function can be of various types, such as an integer or a sequence of integers. When you provide a single integer seed, the seed value is first used to initialize the first element of the state vector. Then, the subsequent elements of the state vector are filled based on a specific algorithm.

Here's a simplified explanation of the initialization process:

1. The seed value is used as the first element of the state vector.

2. A recurrence relation is applied to generate the remaining 623 elements of the state vector. This recurrence relation is defined by the Mersenne Twister algorithm and involves bitwise operations, shifts, and exclusive OR (XOR) operations.

3. After filling the entire state vector, the twist operation is performed. This operation mixes the elements of the state vector to enhance the randomness and improve statistical properties.

4. The Mersenne Twister algorithm requires a warm-up phase to ensure the generated random numbers are not correlated with the initial seed. During this phase, a certain number of random numbers are generated and discarded before the generator is considered fully initialized.

Once the internal state is initialized, subsequent calls to generate random numbers will update and modify the state vector accordingly, producing a sequence of random numbers.

It's worth noting that the specific details of the initialization and the twist operation are more involved and rely on intricate mathematical properties of the Mersenne Twister algorithm. However, this simplified explanation provides a general understanding of how the state vector is initialized in the `std::mt19937` class.

The Xoshiro family of PRNG

The Xoshiro family [7] of Scrambled linear pseudo-random number generators (PRNGs) is a renowned collection of algorithms designed to generate high-quality random numbers with exceptional statistical properties. Developed by Sebastiano Vigna, these PRNGs are widely recognized for their simplicity, speed, and remarkable period lengths.

The Xoshiro family comprises four distinct PRNGs: Xoshiro256**/Xoshiro256++, Xoshiro512**/Xoshiro512++, Xoshiro1024**/Xoshiro1024++, and Xoshiro128+. The nomenclature indicates the state size of each generator, where ++ is a strong summarization scrambler and ** indicates a strong multiplicative scrambler.

Utilizing a combination of bitwise operations, shifts, and xor operations, these PRNGs excel in providing random numbers. They are specifically optimized for 64-bit systems, leveraging the inherent efficiency of native 64-bit arithmetic available on such platforms.

Here's a concise overview of the four members of the Xoshiro family:

1. Xoshiro256**/Xoshiro256++: This PRNG possesses a state size of 256 bits and executes 64 rounds. With a period length of 2^{256} - 1, it generates an extensive array of unique random values before repeating.
2. Xoshiro512**/Xoshiro512++: Featuring a state size of 512 bits and 64 rounds, Xoshiro512** offers an even lengthier period of 2^{512} - 1, ensuring an exceptionally vast sequence of random numbers.
3. Xoshiro1024**/Xoshiro1024++: With a state size of 1,024 bits and 64 rounds, Xoshiro1024** caters to applications that necessitate an extensive stream of random numbers. It boasts an impressive period length of 2^{1024} - 1.
4. Xoshiro128+: Serving as a smaller variant within the Xoshiro family, Xoshiro128+ possesses a state size of 128 bits and executes 24 rounds. Although it has a relatively shorter period length of 2^{128} - 1, it still delivers excellent random number generation capabilities, particularly for applications that do not demand an exceedingly long period.

The Xoshiro family stands as a highly regarded choice due to its exceptional speed, statistical quality, and user-friendliness. These PRNGs find utility in a wide range of applications, including simulations, cryptography, gaming, and general-purpose random number generation.

It is crucial to acknowledge that while the Xoshiro PRNGs demonstrate remarkable efficiency and produce high-quality random numbers, they are not suitable for cryptographic purposes that demand robust security measures.

Source code for Xoshiro256** class

```
// xoshiro256** random number generator implementation
/* This is xoshiro256** 1.0, one of our all-purpose, rock-solid
   generators. It has excellent (sub-ns) speed, a state (256 bits) that is
   large enough for any parallel application, and it passes all tests we
   are aware of.

   For generating just floating-point numbers, xoshiro256+ is even faster.

   The state must be seeded so that it is not everywhere zero. If you have
   a 64-bit seed, we suggest seeding a splitmix64 generator and using its
   output to fill s.
*/
class xoshiro256ss
{// Private
    using result_type = uint64_t;
    std::array<result_type,4> s;   // Internal state 256 bits

    static inline result_type rotl(const result_type x, const int k)
    {
        return (x << k) | (x >> (64 - k));
    }

    static inline result_type splitmix64(result_type x)
    {
        x += 0x9E3779B97F4A7C15;
        x = (x ^ (x >> 30)) * 0xBF58476D1CE4E5B9;
        x = (x ^ (x >> 27)) * 0x94D049BB133111EB;
        return x ^ (x >> 31);
    }

public:
    xoshiro256ss(const result_type val = std::random_device{}() )
    {// Initialization
        seed(val);
    }

    xoshiro256ss(const seed_seq& seeds)
    {   // Initialization through seed_seq seed
        seed(seeds);
    }

    void seed(const result_type seed_value)
    {
        for (int i = 0; i < 4; ++i)
            s[i] = splitmix64(seed_value + i);
    }

   void seed(const seed_seq& seeds)
    {   // Initialization through seed_seq seed
        std::array<unsigned, 4> sequence;
        seeds.generate(sequence.begin(), sequence.end());
        for (int i = 0; i < sequence.size(); ++i)
            s[i] = splitmix64(static_cast<result_type>(sequence[i]));
    }

    static result_type min()
    {
        return result_type(0ull);
    }

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

    result_type operator()()
    {   // 256**
        const uint64_t result = rotl(s[1] * 5, 7) * 9;
        const uint64_t t = s[1] << 17;

        s[2] ^= s[0];
        s[3] ^= s[1];
        s[1] ^= s[2];
        s[0] ^= s[3];
        s[2] ^= t;
        s[3] = rotl(s[3], 45);

        return result;
    }

    bool operator==(const xoshiro256ss& rhs) const
    {
        return this->s == rhs.s;
    }

    bool operator!=(const xoshiro256ss& rhs) const
    {
        return this->s != rhs.s;
    }

    void discard(const unsigned long long z)
    {
        for (unsigned long long i = z; i > 0; --i)
            (void)this->operator()();
    }
};
```

The source for Xoshiro256++, Xoshiro512++, Xoshiro512** can be found in Appendix

The ChaCha family of PRNG.

The ChaCha family of pseudo-random number generators (PRNGs) encompasses a set of stream ciphers devised by Daniel J. Bernstein [5] & [6]. Initially developed for cryptographic purposes, the ChaCha algorithms have also proven valuable in generating random numbers for non-cryptographic applications.

The foundation of the ChaCha family lies in a construction known as a "quarter round" operation, which manipulates a 4x4 matrix of integers. Through the repetitive application of this quarter-round operation in a specific pattern, the ChaCha algorithm generates a stream of pseudo-random numbers.

Within the ChaCha family, various variants exist, including ChaCha8, ChaCha12, and ChaCha20, denoting the number of rounds executed during the quarter-round operation. Typically, a higher number of rounds enhances both the security and the distribution quality of the output, albeit at the cost of increased computational resources.

Each variant offers distinct trade-offs in terms of security and performance, allowing users to select the most suitable one for their specific requirements. Notably, ChaCha20 has gained significant popularity due to its robust security properties and commendable performance.

The ChaCha algorithms are known for their simplicity, high-speed operation, and resilience against cryptographic attacks. They exhibit excellent statistical properties, featuring long periods and uniform distribution of generated random numbers. Consequently, they find applicability across a wide range of domains, encompassing both cryptography and non-cryptographic fields.

Moreover, the ChaCha family extends its utility beyond PRNGs, encompassing symmetric encryption and authentication schemes.

The ChaCha family of PRNGs provides a dependable and efficient solution for generating pseudo-random numbers, with ChaCha20 standing out due to its strong security properties and widespread adoption.

Initialization of the ChaCha20 PRNG involves three distinct keys:

1. key
2. nonce
3. counter

When employing the `ChaCha20` class, it is crucial to select appropriate values for the key, nonce, and counter to ensure the security and uniqueness of the generated pseudo-random numbers. Here are some recommendations:

The key should be a securely generated random sequence of bytes. It is vital to employ a strong and unpredictable key to maintain the security of the ChaCha20 cipher. The key's length should be 256 bits (32 bytes) for ChaCha20.

The nonce (number used once) should possess a unique value for each encryption session and must not be reused with the same key. For ChaCha20, the recommended nonce length is 96 bits (12 bytes). It can be generated randomly or incremented for each session.

The counter serves as an arbitrary value used to differentiate the generated blocks during the pseudo-random number generation process. Typically, it is initialized to 0 and incremented for each new block generated. If multiple instances of the `ChaCha20` class are used concurrently, ensuring unique counters is crucial to avoid collisions.

Here's an example showcasing how to set the key, nonce, and counter when declaring the `ChaCha20` class:

~~~
…
std::vector<uint8_t> key = { /* Your 32-byte key here */ };
std::vector<uint8_t> nonce = { /* Your 12-byte nonce here */ };
uint32_t counter = 0;
ChaCha20 prng(key, nonce, counter);
```
~~~

Remember to utilize suitable random values for the key and nonce, while ensuring that the counter remains unique for each PRNG instance or session.

Furthermore, if ChaCha20 is employed for cryptographic purposes, adhering to best practices and security guidelines is essential. This encompasses proper key management, nonce handling, and overall system security.

Is ChaCha20 considered cryptographic-grade?

ChaCha20 is widely regarded as suitable for cryptographic applications. It serves as a widely employed stream cipher and has been adopted as one of the standard algorithms in various cryptographic protocols and systems.

ChaCha20 offers several advantages that make it well-suited for cryptographic usage:

a. *Security*: Extensive security analysis has confirmed ChaCha20's resilience against known cryptographic attacks. It ensures a high level of confidentiality when deployed as a symmetric encryption algorithm.
b. *Speed and efficiency*: ChaCha20 is designed with an emphasis on speed and efficiency, resulting in highly efficient software implementations. This makes it a preferred choice for resource-constrained devices or applications where performance is critical.
c. *Non-linearity and diffusion*: ChaCha20 incorporates a series of operations, including the quarter round and mixing operations, to achieve strong non-linearity and diffusion properties. These properties ensure that modifications in the input significantly impact the output, bolstering its resistance to cryptographic attacks.
d. *Key flexibility*: ChaCha20 supports key sizes of 128 bits and 256 bits, offering flexibility in selecting the appropriate key length based on the desired security level. Additionally, it employs a 96-bit nonce, enabling the generation of a large number of unique streams.
e. *Widespread support*: ChaCha20 enjoys widespread support in cryptographic libraries and protocols, including TLS/SSL, IPsec, and secure messaging applications.

The ChaCha20 stands as a secure and efficient choice for numerous cryptographic applications, including symmetric encryption, authenticated encryption, and secure communication protocols.

However, it is crucial to follow recommended practices, ensuring proper implementation within the context of the specific cryptographic system or protocol being utilized.

Source code for Chacha20 class

```
// ChaCha20 PRNG class
class chacha20
{
    // ChaCh20 output 32-bit unsigned integers
    // The three initialization key, nonce & counter
    std::vector<uint8_t> key_;
    std::vector<uint8_t> nonce_;
    uint32_t counter_;          // Number of 16 block generated
    std::array<uint32_t, 16> block_; // Holds the next 16 random numbers
    int position_;              // Current position into the block generated

    // ChaCha20 constants
    const std::array<uint32_t, 4> kInitialState = { 0x61707865, 0x3320646e, 0x79622d32,
0x6b206574 };
    const std::array<uint8_t, 16> kSigma = { 'e', 'x', 'p', 'a', 'n', 'd', ' ', '3',
'2', '-', 'b', 'y', 't', 'e', ' ', 'k' };

    // ChaCha20 quarter round operation
    static inline void QuarterRound(uint32_t& a, uint32_t& b, uint32_t& c, uint32_t& d)
    {
        a += b; d ^= a; d = (d << 16) | (d >> 16);
        c += d; b ^= c; b = (b << 12) | (b >> 20);
        a += b; d ^= a; d = (d << 8) | (d >> 24);
        c += d; b ^= c; b = (b << 7) | (b >> 25);
    }

    // ChaCha20 core function
    static void ChaCha20Core(const std::array<uint32_t, 16>& input, std::array<uint32_t,
16>& output)
    {
        std::array<uint32_t, 16> state = input;

        for (int i = 0; i < 10; ++i) {
            // Column rounds
            QuarterRound(state[0], state[4], state[8], state[12]);
            QuarterRound(state[1], state[5], state[9], state[13]);
            QuarterRound(state[2], state[6], state[10], state[14]);
            QuarterRound(state[3], state[7], state[11], state[15]);

            // Diagonal rounds
            QuarterRound(state[0], state[5], state[10], state[15]);
            QuarterRound(state[1], state[6], state[11], state[12]);
            QuarterRound(state[2], state[7], state[8], state[13]);
            QuarterRound(state[3], state[4], state[9], state[14]);
        }

        for (int i = 0; i < 16; ++i) {
            output[i] = state[i] + input[i];
        }
    }

    // Generate the next 16 random numbers
    void generateNewBlock()
    {
        std::array<uint32_t, 16> input;
        std::array<uint32_t, 16> output;
        // Set the ChaCha20 initial state
        input[0] = kInitialState[0];
        input[1] = kInitialState[1];
        input[2] = kInitialState[2];
        input[3] = kInitialState[3];

        // Set the key, nonce, and counter
        std::copy(kSigma.begin(), kSigma.end(), reinterpret_cast<uint8_t*>(&input[4]));
        std::copy(key_.begin(), key_.end(), reinterpret_cast<uint8_t*>(&input[8]));
        std::copy(nonce_.begin(), nonce_.end(), reinterpret_cast<uint8_t*>(&input[12]));
        input[14] = counter_;

        ChaCha20Core(input, output);

        // Copy the output to the block
        std::copy(output.begin(), output.end(), block_.begin());
        ++counter_;
    }

public:
    // Constructor
    chacha20(const std::vector<uint8_t>& key, const std::vector<uint8_t>& nonce,
uint32_t counter)
        : key_(key), nonce_(nonce), counter_(counter), position_(0) {}
    chacha20()
    {
        seed();
    }

    // Seed
    void seed(const std::vector<uint8_t>& key, const std::vector<uint8_t>& nonce,
uint32_t counter)
    {
        key_ = key;
        nonce_ = nonce;
        counter_ = counter;
        position_ = 0;
    }

    // Seed with value
    void seed(const uint32_t s= std::random_device{}())
    {
        mt19937 gen(s);  // use the build in mt19937 PRNG for random values
        uniform_int_distribution<uint32_t> dis(1, 0xfe);

        key_.clear();
        for (int i = 0; i < 16; ++i)
            key_.push_back(static_cast<uint8_t>(dis(gen)));
        nonce_.clear();
        for (int i = 0; i < 8; ++i)
            nonce_.push_back(static_cast<uint8_t>(dis(gen)));
        counter_ = gen();
        position_ = 0;
    }

    void seed(const std::seed_seq& seeds)
    {// Initialization through seed_seq seed
        std::array<uint32_t, 16> sequencekey;
        std::array<uint32_t, 16> sequencenonce;
        std::array<uint32_t, 1> sequencecounter;
        seeds.generate(sequencekey.begin(), sequencekey.end());
        key_.clear();
        for (int i = 0; i < 16; ++i)
            key_.push_back(static_cast<uint8_t>(sequencekey[i]));
        seeds.generate(sequencenonce.begin(), sequencenonce.end());
        nonce_.clear();
        for (int i = 0; i < 8; ++i)
            nonce_.push_back(static_cast<uint8_t>(sequencenonce[i]));

        seeds.generate(sequencecounter.begin(), sequencecounter.end());
        counter_ = sequencecounter[0];
        position_ = 0;
    }

    uint32_t operator()()
    {
        if (position_ == 0 || position_ >= 16)
        {
            generateNewBlock();
            position_ = 0;
        }

       uint32_t randomNumber = block_[position_];
        ++position_;
        return randomNumber;
    }

    static constexpr uint32_t min()
    {
        return uint32_t(0ul);
    }

    static constexpr uint32_t max()
    {
        return std::numeric_limits<uint32_t>::max();
    }

    bool operator==(const chacha20& rhs) const
    {
        return this->key_ == rhs.key_ && this->nonce_ == rhs.nonce_ && this->counter_ ==
rhs.counter_;
    }

    bool operator!=(const chacha20& rhs) const
    {
        return this->key_ != rhs.key_ || this->nonce_ != rhs.nonce_ || this->counter_ !=
rhs.counter_;
    }

    void discard(const unsigned long z)
    {
        for (unsigned long long i = z; i > 0; --i)
            (void)this->operator()();
    }
};
```

The arbitrary precision version of a template-based PRNG.

We have previously laid out the requirement for an arbitrary precision version of a PRNG. It will follow the same design principles as in the C++ built-in PRNGs library. Since the arbitrary previous version will utilize the core PRNGs in the C library plus the Xoshiro and the Chacha20 mentioned in this paper. We can then define a C++ template version of a class called random_precision. The template version has two parameters. The first parameter is the PRNG class used in the C++ library or the Xoshiro and the ChaCha20 presented in this document. The second parameter is the type of output from the core random generators. This is either uint32_t (32-bit) or uint64_t (64-bit) to match the chosen PRNG core class.

Below is a list of available PRNG class types that can be used in connection with the arbitrary precision version of the PRNG.

Type name	Family	Return_type
minstd_rand	Linear congruential generator	32-bit
mt19937	Mersenne twister	32-bit
mt19937_64	Mersenne twister	64-bit
ranlux24	Subtract and carry	32-bit
ranlux48	Subtract and carry	64-bit
knuth_b	Shuffle linear congruential generator	32-bit
default_random_engine	Any of the above (implementation-defined)	32-bit or 64-bit*
rand()	Linear congruential generator	32-bit
Xoshiro family	Scramble linear	64-bit
Chacha20	“quarter round”	32-bit

*) implementation dependent.

Here are a few examples of how to declare the arbitrary precision version.

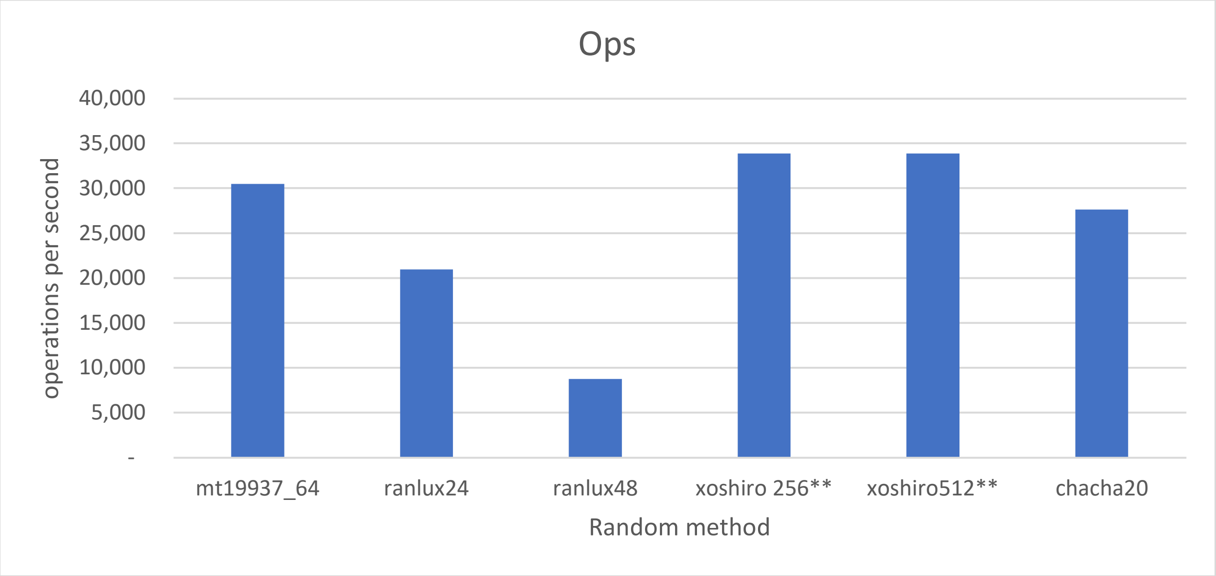
```
random_precision<mt19937_64> genmt19937_64bit;
random_precision<mt19937,uint32_t> genmt19937_32bit;
random_precision<ranlux24,uint32_t> genranlux24_32bit;
random_precision<ranlux48,uint64_t> genranlux48_64bit;
```

Source for the Random_precision template class

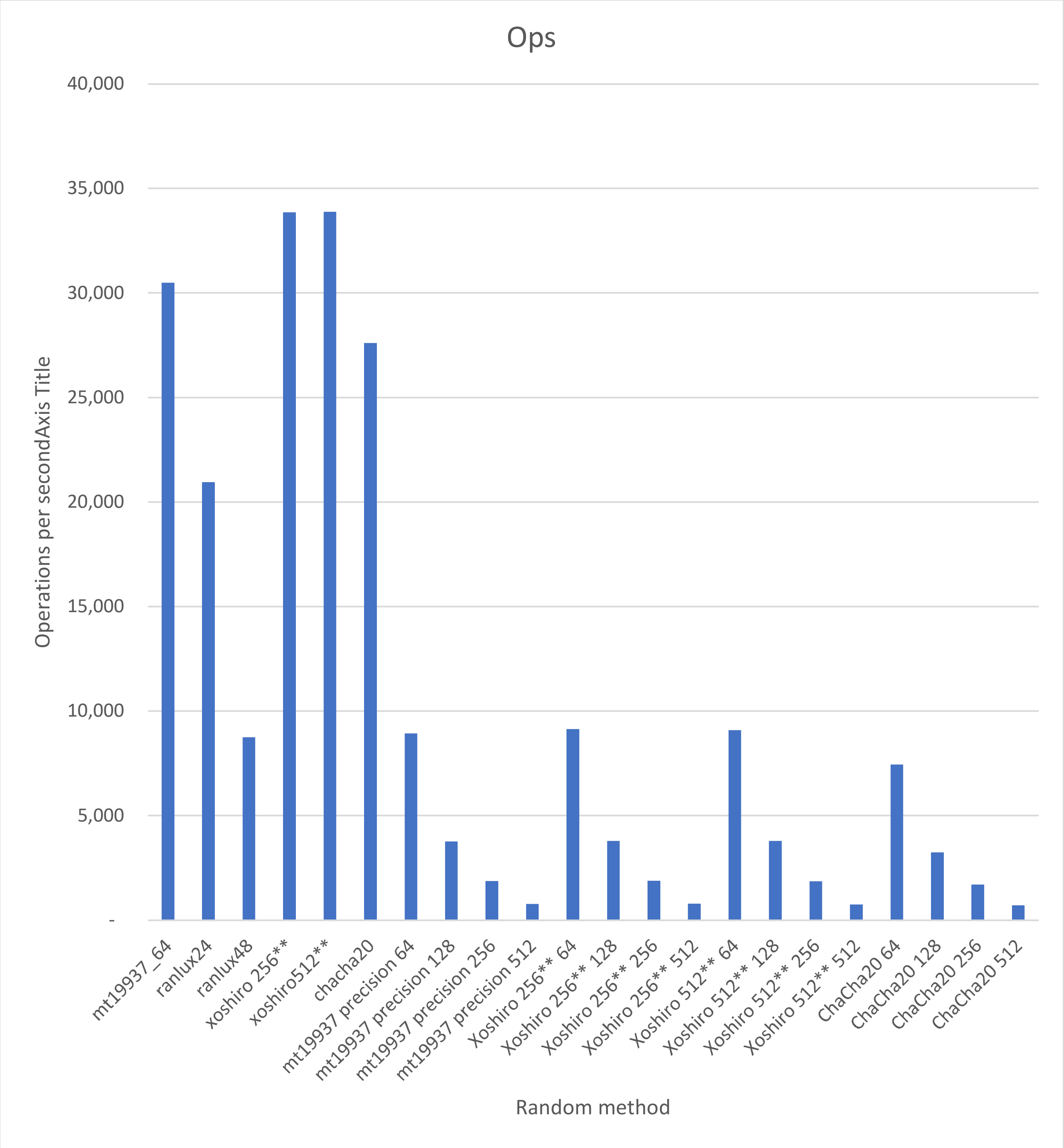
```
template<class _prng, class _rettype = uint64_t> class random_precision
{
    using result_type = int_precision;
    _prng generator;

    static inline unsigned long long splitmix64(unsigned long long x) {
        x += 0x9E3779B97F4A7C15;
        x = (x ^ (x >> 30)) * 0xBF58476D1CE4E5B9;
        x = (x ^ (x >> 27)) * 0x94D049BB133111EB;
        return x ^ (x >> 31);
    }

public:
    random_precision(const result_type val = std::random_device{}())
    {   // Initialization
        seed(val);
    }

    random_precision(const std::seed_seq& seeds)
    {   // Initialization through seed_seq seed
        seed(seeds);
    }

    void seed(const result_type seed_value)
    {   // Seed value is either a uint32_t or uint64_t
        generator.seed(_rettype(seed_value));
    }

    void seed(const std::seed_seq& seeds)
    {   // Initialization through seed_seq seed
        std::array<unsigned, 1> sequence;
        seeds.generate(sequence.begin(), sequence.end());
        generator.seed(splitmix64(static_cast<unsigned long long>(sequence[0])));
    }

    static result_type min()
    {
        return result_type(0ull);
    }

    static result_type max(const uintmax_t bitcnt = 64)
    {
        if (bitcnt <= 64)
            return result_type(~0ull);
        int_precision m;
        m.setbit(bitcnt);           // 2^bitcnt
        m -= int_precision(1);      // 2^bitcnt-1
        return m;
    }

    result_type operator()(const uintmax_t bitcnt = 64)
    {
        result_type result(0);
        iptype a;
        size_t bcnt = bitcnt % 64;
        // Ensure uniform distribution of the random numbers
        uniform_int_distribution<uintmax_t> disbits(0, bitcnt);
        uniform_int_distribution<uintmax_t> dis(0, (~0ull));  // Full 64bit range

        bcnt = disbits(generator);
        a = bcnt % 64;
        // Build int_precision random number
        // Set most significant 64bits segment
        if (a != 0)
        {
            if (bcnt < 64)
                a = dis(generator);
            else
            {
                if (a == 63)
                    a = ~(0ull);
                else
                {
                    a = 1ull << (a + 1);
                    a -= 1;
                }
                uniform_int_distribution<uintmax_t> distop(0, a);      // Most 
significant range
                a = distop(generator);
            }
            result = int_precision(a);
        }
        for (; bcnt > 64; bcnt -= 64)
        {
            result <<= Bitsiptype;
            result += dis(generator);
        }

        return result;
    }

    bool operator==(const random_precision& rhs) const
    {
        return this->generator == rhs.generator;
    }

    bool operator!=(const random_precision& rhs) const
    {
        return this->generator != rhs.generator;
    }

    void discard(const unsigned long long z)
    {
        for (unsigned long long i = z; i > 0; --i)
            (void)this->operator()();
    }
};
```

Performance

Testing of the various random methods. We notice that ranlux24 and ranlux48 indeed are slower and considerably slower than any of the other methods. Both Xoshiro methods are slightly faster than mt19937 and ChaCha20 is trailing the mt199937 but only by a small amount. If speed is of the essence, then I recommend the Xoshiro256 and Xoshiro512 methods. If you favor the cryptographic graded method then I recommend the ChaCha20. Although ranlux48 is a PRNG with high quality it is far behind the other methods in terms of speed. Generally, I agreed with [2] & [3] that there is no need to consider ranlux24 or ranlux48 unless you don’t have an implementation of the Xoshiro or ChaCha20



If you look at the performance of the arbitrary precision versions of mt19937, the Xoshiro family, and the ChaCha20 we see a similar picture. The numbers 64, 128, 256, and 512 behind the name of the arbitrary precision version indicate the output in bits. E.g. the Xoshiro 256** 512 is the Xoshiro256** version delivering a random number from 64bits to the size of 512bits in this performance test.



We see here that the arbitrary precision version is close to 4 times slower than the “native” counterpart. This can be viewed as the cost of computing arbitrary precision integers. And as we required larger random arbitrary precision integers the performance fell by a little more than a factor of two as we double the output of the size of the arbitrary precision integer.

Recommendation

For a final recommendation, I recommend one of the Xoshiro family if speed is of the essence and the ChaCha20 if cryptographic grade quality is needed.

Reference

1) Arbitrary precision library package. Arbitrary Precision C++ Packages
2) Learn cpp chapter 7.19. 7.19 — Introduction to random number generation – Learn C++ (learncpp.com)
3) Learn cpp chapter 7.20. 7.20 — Generating random numbers using Mersenne Twister – Learn C++ (learncpp.com)
4) ChatGPT (www.openai.com) on March 5, 2023
5) D.J. Bernstein. ChaCha. The ChaCha family of stream ciphers (yp.to)
6) D.J. Bernstein. ChaCha is a variant of Salsa. chacha-20080128.pdf (yp.to)
7) Xoshiro family of PRNGs. https://prng.di.unimi.it/

Appendix

Source Xoshiro Class

Source for Xoshiro256++ class

```
// xoshiro256++ random number generator implementation
/* This is xoshiro256++ 1.0, one of our all-purpose, rock-solid generators.
   It has excellent (sub-ns) speed, a state (256 bits) that is large
   enough for any parallel application, and it passes all tests we are
   aware of.

   For generating just floating-point numbers, xoshiro256+ is even faster.

   The state must be seeded so that it is not everywhere zero. If you have
   a 64-bit seed, we suggest seeding a splitmix64 generator and using its
   output to fill s. */
class xoshiro256pp {
    using result_type = uint64_t;
    std::array<result_type, 4> s;   // Internal state 256 bits

    static inline result_type rotl(const result_type x, const int k)
    {
        return (x << k) | (x >> (64 - k));
    }

    static inline result_type splitmix64(result_type x)
    {
        x += 0x9E3779B97F4A7C15;
        x = (x ^ (x >> 30)) * 0xBF58476D1CE4E5B9;
        x = (x ^ (x >> 27)) * 0x94D049BB133111EB;
        return x ^ (x >> 31);
    }

public:
    xoshiro256pp(const result_type val = std::random_device{}() /* 5489u*/)
    {// Initialization
        seed(val);
    }

    xoshiro256pp(const seed_seq& seeds)
    {   // Initialization through seed_seq seed
        seed(seeds);
    }

    void seed(const result_type seed_value)
    {
        for (int i = 0; i < 4; ++i)
            s[i] = splitmix64(seed_value + i);
    }

   void seed(const seed_seq& seeds)
    {   // Initialization through seed_seq seed
        std::array<unsigned, 4> sequence;
        seeds.generate(sequence.begin(), sequence.end());
        for (int i = 0; i < 4; ++i)
            s[i] = splitmix64(static_cast<result_type>(sequence[i]));
    }

    static result_type min()
    {
        return result_type(0u);
    }

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

    result_type operator()()
    {  /// 256++
        const result_type result = rotl(s[0] + s[3], 23) + s[0];
        const result_type t = s[1] << 17;

        s[2] ^= s[0];
        s[3] ^= s[1];
        s[1] ^= s[2];
        s[0] ^= s[3];
        s[2] ^= t;
        s[3] = rotl(s[3], 45);

        return result;
    }

    bool operator==( const xoshiro256pp& rhs) const
    {
        return this->s==rhs.s;
    }

    bool operator!=(const xoshiro256pp& rhs) const
    {
        return this->s != rhs.s;
    }

    void discard(const unsigned long long z)
    {
        for (unsigned long long i = z; i > 0; --i)
            (void)this->operator()();
    }
};
```

Source for Xoshiro512++ class

```
// xoshiro512++ random number generator implementation
/* This is xoshiro512++ 1.0, one of our all-purpose, rock-solid
   generators. It has excellent (about 1ns) speed, a state (512 bits) that
   is large enough for any parallel application, and it passes all tests
   we are aware of.

   For generating just floating-point numbers, xoshiro512+ is even faster.

   The state must be seeded so that it is not everywhere zero. If you have
   a 64-bit seed, we suggest seeding a splitmix64 generator and using its
   output to fill s.
   */
class xoshiro512pp {
    using result_type = uint64_t;
    std::array<result_type,8> s;   // Internal state 512 bits

    static inline result_type rotl(const result_type x, const int k) {
        return (x << k) | (x >> (64 - k));
    }

    static inline result_type splitmix64(result_type x) {
        x += 0x9E3779B97F4A7C15;
        x = (x ^ (x >> 30)) * 0xBF58476D1CE4E5B9;
        x = (x ^ (x >> 27)) * 0x94D049BB133111EB;
        return x ^ (x >> 31);
    }

public:
    xoshiro512pp(const result_type val = std::random_device{}() /*5489u*/)
    {// Initialization
        seed(val);
    }

    xoshiro512pp(const seed_seq& seeds)
    {   // Initialization through seed_seq seed
        seed(seeds);
    }

    void seed(const result_type seed_value) {
        for (int i = 0; i < 8; ++i) {
            s[i] = splitmix64(seed_value + i);
        }
    }

    void seed(const seed_seq& seeds)
    {   // Initialization through seed_seq seed
        std::array<unsigned, 8> sequence;
        seeds.generate(sequence.begin(), sequence.end());
        for (size_t i = 0; i < sequence.size(); ++i)
            s[i] = splitmix64(static_cast<result_type>(sequence[i]));
    }

    static result_type min()
    {
        return result_type(0ull);
    }

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

    result_type operator()()
    {//512++
        const result_type result = rotl(s[0] + s[2], 17) + s[2];
        const result_type t = s[1] << 11;

        s[2] ^= s[0];
        s[5] ^= s[1];
        s[1] ^= s[2];
        s[7] ^= s[3];
        s[3] ^= s[4];
        s[4] ^= s[5];
        s[0] ^= s[6];
        s[6] ^= s[7];
        s[6] ^= t;
        s[7] = rotl(s[7], 21);

        return result;
    }

    bool operator==(const xoshiro512pp& rhs) const
    {
        return this->s == rhs.s;
    }

    bool operator!=(const xoshiro512pp& rhs) const
    {
        return this->s != rhs.s;
    }

    void discard(const unsigned long long z)
    {
        for (unsigned long long i = z; i > 0; --i)
            (void)this->operator()();
    }
};
```

Source code for Xoshiro512** class

```
/* This is xoshiro512** 1.0, one of our all-purpose, rock-solid generators
   with increased state size. It has excellent (about 1ns) speed, a state
   (512 bits) that is large enough for any parallel application, and it
   passes all tests we are aware of.

   For generating just floating-point numbers, xoshiro512+ is even faster.

   The state must be seeded so that it is not everywhere zero. If you have
   a 64-bit seed, we suggest seeding a splitmix64 generator and using its
   output to fill s.
   */
class xoshiro512ss {
    using result_type = uint64_t;
    std::array<result_type,8> s;       // Internal state 512 bits

    static inline result_type rotl(const result_type x, const int k) {
        return (x << k) | (x >> (64 - k));
    }

    static inline result_type splitmix64(result_type x) {
        x += 0x9E3779B97F4A7C15;
        x = (x ^ (x >> 30)) * 0xBF58476D1CE4E5B9;
        x = (x ^ (x >> 27)) * 0x94D049BB133111EB;
        return x ^ (x >> 31);
    }

public:
    xoshiro512ss(const result_type val = std::random_device{}())
    {   // Initialization
        seed(val);
    }

    xoshiro512ss(const seed_seq& seeds)
    {   // Initialization through seed_seq seed
        seed(seeds);
    }

    void seed(const result_type seed_value)
    {   // Regular seed
        for (int i = 0; i < 8; ++i) {
            s[i] = splitmix64(seed_value + i);
        }
    }

    void seed(const seed_seq& seeds)
    {   // Initialization through seed_seq seed
        std::array<unsigned, 8> sequence;
        seeds.generate(sequence.begin(), sequence.end());
        for (int i = 0; i < sequence.size(); ++i)
            s[i] = splitmix64(static_cast<result_type>(sequence[i]));
    }

    static result_type min()
    {
        return result_type(0ull);
    }

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

    result_type operator()()
    {//512**
        const result_type result = rotl(s[1] * 5, 7) * 9;
        const result_type t = s[1] << 11;

        s[2] ^= s[0];
        s[5] ^= s[1];
        s[1] ^= s[2];
        s[7] ^= s[3];
        s[3] ^= s[4];
        s[4] ^= s[5];
        s[0] ^= s[6];
        s[6] ^= s[7];
        s[6] ^= t;
        s[7] = rotl(s[7], 21);

        return result;
    }

    bool operator==( const xoshiro512ss& rhs) const
    {
        return this->s == rhs.s;
    }

    bool operator!=(const xoshiro512ss& rhs) const
    {
        return this->s != rhs.s;
    }

    void discard(const unsigned long long z)
    {
        for (unsigned long long i = z; i > 0; --i)
            (void)this->operator()();
    }
};
```