Random number generator
for C++ template metapograms*

Zalán Szűgyi, Tamás Cséri, and Zoltán Porkoláb

Department of Programming Languages and Compilers, Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
{lupin, cseri, gsd}@caesar.elte.hu

Abstract. Template metaprogramming is a widely used programming paradigm to develop libraries in C++. With the help of cleverly defined templates the programmer can execute algorithms at compilation time. C++ template metaprograms are proven to be Turing-complete, thus wide scale of algorithms can be executed in compilation time. Applying randomized algorithms and data structures is, however, troublesome due to the deterministic nature of template metaprograms. In this paper we describe a C++ template metaprogram library that generates pseudorandom numbers at compile time. Random number engines are responsible to generate pseudorandom integer sequences with a uniform distribution. Random number distributions transform the generated pseudorandom numbers into different statistical distributions. Our goal was to provide similar functionality to the run-time random generator module of the Standard Template Library, thus programmers familiar with STL can easily adopt our library.

1 Introduction

Template metaprogramming is a modern, still developing programming paradigm in C++. It utilizes the instantiation technique of the C++ templates and makes the C++ compiler execute algorithms in compilation time. Template metaprograms were proven to be a Turing-complete sublanguage of C++ [4], which means that a wide set of algorithms can be executed at compile time within the limits of the C++ compiler resources.

We write template metaprograms for various reasons, like expression templates [23] replacing runtime computations with compile-time activities to enhance runtime performance, static interface checking, which increases the ability of the compile-time to check the requirements against template parameters, i.e. they form constraints on template parameters [11,19], active libraries [24], acting dynamically during compile-time, making decisions based on programming contexts and making optimizations. The Ararat system [5], boost::xpressive [13], and boost::proto [14] libraries provide metaprogramming solutions to embed DSLs into C++. Another approach to embed DSLs is to reimplement the Haskell’s parser generators library with C++ template metaprograms [17].

* The project was supported by Ericsson Hungary.
Boost metaprogram library (MPL) \cite{6} provides basic containers, algorithms and iterators to help in basic programmer tasks, similarly to their runtime correspondents in the Standard Template Library (STL). Fundamental types, like string are also exists in MPL, and can be extended with more complex functionality \cite{22}. Since C++ template metaprograms follow the functional paradigm, non-STL-like approaches exists in metaprogram development too: functional programming idioms, like nested lambda and let expressions are also introduced in paper \cite{20}.

Not only libraries, but C++11, the new standard of the C++ programming language also supports template metaprogramming. There are new keywords and language structures such as constant expressions, decltype, variadic templates which makes writing metaprograms easier.

It is common in all the methods, techniques and libraries we discussed that C++ template metaprograms are inheritably deterministic. Algorithms, data types are fully specified by the source code and do not depend on any kind of external input. This means that the same metaprogram will always be executed the same way and will produce the same results (generated code, data types, compiler warnings, etc.) every time.

Therefore, it is very difficult to implement randomized algorithms and data structures with C++ template metaprograms, due to the deterministic behavior described above. Nevertheless, undeterministic algorithms and data structures are important in a certain class of tasks as they are often simpler, and more efficient than their deterministic correspondents.

Finding a minimal cut on an undirected graph is a fundamental algorithm in network theory for partitioning elements in a database, or identifying clusters of related documents. The deterministic algorithm is very complex and difficult \cite{9}, a much smaller and easier algorithm can be written using random choice \cite{8}. A skip list \cite{18} is an alternative to search trees. The rotation methods in search trees are algorithmically complex. Based on random numbers, skip list provides simpler way to reorganize the data structure, which is essential in template metaprograms due to their complex syntax and burdensome debug possibilities. Algorithms that selects pivot elements to partition their input sequence such as the quick sort algorithm and the similar $k^{th}$ minimal element selection algorithm provides better worst case scenarios if we select the pivot element randomly.

In this paper we describe our C++ template metaprogram library that generates pseudorandom numbers at compile time. Random number engines are responsible to generate pseudorandom integer sequences with a uniform distribution. Random number distributions transform the generated pseudorandom numbers into different statistical distributions. Engines and distributions can be used together to generate random values. The engines are created using user defined seeds, allowing to generate repeatable random number sequences.

Our goal was to design our library similar to the runtime random library provided by the STL. Thus a programmer familiar with the STL can easily adopt our library to their metaprograms.
Our paper organizes as follows: In Section 2 we discuss those C++ template metaprogramming constructs which form the implementation base of our library. Section 3 introduces our compile time random number generator library with implementational details. In Section 4 we show how our library can be applied for real life problems and we evaluate the results. Section 5 mentions a project related to code obfuscation using some randomization in C++ template metaprograms. Future works are discussed in Section 6. Our paper concludes in Section 7.

2 Template metaprogramming

The template facilities of C++ allow writing algorithms and data structures parametrized by types. This abstraction is useful for designing general algorithms like finding an element in a list. The operations on lists of integers, characters or even user defined classes are essentially the same. The only difference between them is the stored type. With templates we can parametrize these list operations by abstract type, thus, we need to write the abstract algorithm only once. The compiler will generate the integer, double, character or user defined class version of the list by replacing the abstract type with a concrete one. This method is called instantiation.

The template mechanism of C++ enables the definition of partial and full specializations. Let us suppose that we would like to create a more space efficient type-specific implementation of the list template for bool type. We may define the following specialization:

```cpp
template<typename T>
struct list
{
  void insert(const T& e);
  /* ... */
};

template<>
struct list<bool>
{
  //type-specific implementation
  void insert(bool e);
  /* ... */
};
```

Programs that are evaluated at compilation time are called metaprosgrams. C++ supports metaprogramming via preprocessor macros and templates. Preprocessor macros run before the C++ compilation and therefore they are unaware of the C++ language semantics. However, template metaprograms are evaluated during the C++ compilation phase, therefore the type safety of the language is enforced.
Template specialization is essential practice for template metaprogramming [1]. In template metaprograms templates usually refer to themselves with different type arguments. Such chains of recursive instantiations can be terminated by a template specialization. See the following example of calculating the factorial value of 5:

```cpp
template<int N>
struct factorial
{
    enum { value = N * factorial<N-1>::value }; 
};

template<> 
struct factorial<0>
{
    enum { value = 1 }; 
};

int main()
{
    int result = factorial<5>::value;
}
```

To initialize the variable `result`, the expression `factorial<5>::value` has to be evaluated. As the template argument is not zero, the compiler instantiates the general version of the `factorial` template with 5. The definition of `value` is `N * factorial<N-1>::value`, hence the compiler has to instantiate the `factorial` again with 4. This chain continues until the concrete value becomes 0. Then, the compiler chooses the special version of `factorial` where the `value` is 1. Thus, the instantiation chain is stopped and the factorial of 5 is calculated and used as initial value of the `result` variable in `main`. This metaprogram “runs” while the compiler compiles the code.

Template metaprograms therefore consist of a collection of templates, their instantiations and specializations, and perform operations at compilation time. Basic control structures like iterations and conditions are represented in a functional way [21]. As we can see in the previous example, iterations in metaprograms are applied by recursion. Besides, the condition is implemented by a template structure and its specialization.

```cpp
template<bool cond_, typename then_, typename else_>
struct if_
{
    typedef then_ type;
};

template<typename then_, typename else_>
```
struct if_<false, then_, else_>
{
    typedef else_ type;
};

The if_ structure has three template arguments: a boolean and two abstract types. If the cond_ is false, then the partly-specialized version of if_ will be instantiated, thus the type will be bound by the else_. Otherwise the general version of if_ will be instantiated and type will be bound by then_.

Complex data structures are also available for metaprograms. Recursive templates store information in various forms, most frequently as tree structures, or sequences. Tree structures are the most common forms of implementation of expression templates [23]. The canonical examples for sequential data structures are Typelist [2] and the elements of the boost::mpl library [6].

We define a type list with the following recursive template:

class NullType {};
struct EmptyType {}; // could be instantiated

template <typename H, typename T>
struct Typelist
{
    typedef H head;
    typedef T tail;
};
typedef Typelist< char, Typelist<signed char,
        Typelist<unsigned char, NullType> > > Charlist;

In the example we store the three character types in our Typelist. We can use helper macro definitions to make the syntax more readable.

#define TYPELIST_1(x) Typelist< x, NullType>
#define TYPELIST_2(x, y) Typelist< x, TYPELIST_1(y)>
#define TYPELIST_3(x, y, z) Typelist< x, TYPELIST_2(y,z)>
// ...
typedef TYPELIST_3(char, signed char, unsigned char) Charlist;

Essential helper functions – like Length, which computes the size of a list at compilation time – have been defined in Alexandrescu’s Loki library[2] in pure functional programming style. Similar data structures and algorithms can be found in the boost::mpl metaprogramming library. The Boost Metaprogramming Library [6] is a general-purpose, high-level C++ template metaprogramming framework of algorithms, sequences and metafunctions. The architecture is similar to the Standard Template Library (STL) of C++ with containers, algorithms and iterators but boost::mpl offers this functionality in compilation time.
3 Compile-time random number generation

Our random number generator library for template metaprograms is designed to be similar to the runtime random number library provided by Standard Template Library (STL) of the new standard of C++. Our library provides:

- **random number engines**, that generate pseudorandom integer sequences with uniform distribution.
- **random number distributions**, that transform the output of the random number engines into different statistical distributions.

Engines and distributions can be used together to generate random values. The engines are created using user defined seeds, allowing to generate repeatable random number sequences.

3.1 Basic metafunctions

Before we detail our engines and distributions, we present some basic metafunctions which are commonly used in our implementation. The `Random` metafunction initializes a random engine or distribution, and returns with the first random number in a sequence. See the code below:

```cpp
template<typename Engine>
struct Random
{
  typedef typename init<Engine>::type type;
  static const decltype(type::value) value = type::value;
};
```

The initialization is done by the `init` metafunction, which is partially specialized for all engines and distributions. The first random number is stored in static field `value`.

The `Next` metafunction computes the next random number in a sequence. See its code below:

```cpp
template<typename R>
struct Next
{
  typedef typename eval<R>::type type;
  static const decltype(type::value) value = type::value;
};
```

The next element is computed by the `eval` metafunction, which is, similarly to `init`, partially specialized for all engines and distributions. The return value of this metafunction is stored in static field `value`.
3.2 Random number engines

Similarly to STL, we implemented three random number engines:

- *linear congruential engine* [15], which requires very small space to store its state, and moderately fast.
- *subtract with carry engine* [3], which produces a better random sequence, very fast, but requires more state storage.
- *Mersenne twister engine* [10], which is slower and has even more state storage requirements but with the right parameters provides the longest non-repeating sequence of random numbers.

The implementation of these engines contain three major entities. The first one is a metatype, that contains the state of the engine. This metatype is the argument of the partially specialized `init` and `eval` metafunctions, which do some initialization steps and evaluate the next random number, respectively.

For the *linear congruential engine*, these entities defined as below:

```cpp
template<typename UIntType,
    UIntType seed = defaultseed,
    UIntType a = 16807,
   UIntType c = 0,
    UIntType m = 2147483647>
struct linear_congruential_engine
{
    static const UIntType value = seed;
    static const UIntType maxvalue = m-1;
};
```

The first template argument specifies the type of the random numbers. The type can be any unsigned integer type. The second argument is the random seed. This is an optional parameter. If the programmer does not specify it, an automatically generated random seed is applied for each compilation. See Subsection 3.4 for more details. The other arguments are parameters of the linear congruential equation.

The `init` and `eval` metafunctions can be seen below:

```cpp
template<
    typename UIntType,
    UIntType seed,
    UIntType a,
    UIntType c,
    UIntType m>
struct init<linear_congruential_engine<UIntType, seed, a, c, m>>
{
    typedef typename eval<linear_congruential_engine<
        UIntType, seed, a, c, m>>::type type;
    static const UIntType value = type::value;
};
```
The metafunction \texttt{eval} computes the next random number, stores it in the static field \texttt{value} and modifies the state of the \textit{linear congruential engine}. Metafunction \texttt{init} just invokes \texttt{eval} to compute the first random number.

\subsection*{3.3 Random number distributions}

A \textit{random number distribution} transforms the uniformly distributed output of a random number engine into a specific statistical distribution. Although the STL defines discrete and continuous distributions, we implement only the discrete ones in our library, because of the lack of support of floating point numbers in the template facility of C++. However, we plan to extend the template system with rational and floating point number types that we can use to implement continuous distributions. Our library implements the following discrete probability distributions: \textit{uniform integer distribution}, \textit{Bernoulli-distribution}, \textit{binomial distribution}, \textit{negative binomial distribution}, \textit{geometric distribution}, \textit{Poisson-distribution}, \textit{discrete distribution} \cite{7}. Several distributions require a real number as argument. As the template system in C++ does not accept floating point numbers, our library deals with these parameters as rational numbers: receives the numerator and the denominator as integers. The compiler can approximate the quotient inside the metafunctions. See the implementation of Bernoulli distribution below:

\begin{verbatim}
template<
    typename Engine,
    int N, int D,
    bool val = false
>
struct Bernoulli
{
    static const bool value = val;
};

template<
    typename Engine,
    int N, int D,
    bool b
>
struct eval<Bernoulli<Engine, N, D, b>>
{

}
\end{verbatim}
typedef typename Next<Engine>::type tmptype;
static const bool value =
    (static_cast<double>(tmptype::value) / tmptype::maxvalue) <
    (static_cast<double>(N) / D);
typedef Bernoulli<tmptype, N, D, value> type;
};

The template parameter Engine refers to any kind of random number engine. The integers N and D represent the parameter of Bernoulli distributions, and val stores the computed result. The partially specialized eval metafunction transforms the result of the engine to a boolean value according to the parameter and sets the new state of the Bernoulli class. The other distributions are implemented in similar way.

3.4 Random seed

Pseudorandom number generators require an initial state, called random seed. The random seed determines the generated random sequence. The random number engines in our library optionally accept seed. Specifying the seed results in reproducible sequences of random numbers. However, if the seed is omitted, the library will generate a random seed based on the current time of the system. The time is received using the __TIME__ macro, which is preprocessed to a constexpr character array. Our library will compute a seed using the elements of this array. See the code below:

constexpr char inits[] = __TIME__;
const int defaultseed = (inits[0] - '0')*100000 +
    (inits[1] - '0')*10000 +
    (inits[3] - '0')*1000 +

The C++ standard defines that the preprocessor of C++ should translate the __TIME__ macro to a character string literal in "hh:mm:ss" form. We transform it into a six-digit integer, excluding the colons.

3.5 Example

In this subsection we show a basic usage of our library. We print ten boolean values having Bernoulli distribution with parameter 0.1. We use the linear congruential engine to generate the random sequence.
template<int cnt, typename R>
struct print_randoms
{
    static void print()
    {
        typedef typename Next<R>::type RND;
        std::cout << RND::value << " ";
        print_randoms<cnt-1, RND >::print();
    }
};

template<typename R>
struct print_randoms<0, R>
{
    static void print()
    {
        std::cout << Next<R>::value << " " << std::endl;
    }
};

int main()
{
    print_randoms<10,
        typename Random<
            Bernoulli<
                linear_congruential_engine<
                    uint_fast32_t>,
                    1,
                    10
                >
            >::type
        >::print();
}

4 Evaluation

The Boost MPL uses deterministic algorithms in its implementation. For example the boost::mpl::sort metafunction always selects the first element of the current range as its pivot element. This strategy leads to worst-case scenario when the data is already sorted. It has also a performance overhead if parts of the input data are sorted. However, if we select the pivot element randomly, the worst-case scenario will occur on least common patterns.

We combined the sort algorithm with our library: we chose the pivot element randomly. We evaluated both methods with random and ordered data. Experiments show that we achieved great speedup on the ordered input sam-
ple, whereas the performance loss on the random sample is minor. (Figure 1) Compiling the templates requires not only CPU power, but the instantiations must be stored in the memory as well. Because the random algorithm requires less instantiation steps, we also need less memory for the compilation. (Figure 2)

One fundamental strategy lacking from the current implementation of our library is splitting the random number generator. Metaprograms are written in functional programming style and therefore it is not possible to use a global random generator variable. This shortcoming can be overcome if the random sequencer supports splitting.

Note that the lack of splitting is not a problem in the quicksort algorithm as the two parts of the sequence become independent after separating the sequence based on the pivot element and therefore the same random numbers provide as much randomness as different ones. Therefore, this technique here is rather an optimization as it requires less template instantiation steps.

5 Related work

Neves et. al. developed a code obfuscator library for C++ programming language [12]. The strength of this library is that the obfuscation steps are template metaprograms, thus the programmer does not need to deal with obfuscation, this process is done automatically by the C++ compiler during code compilation. They used randomness to avoid the usage of the same transformation repeatedly. The implemented a very simple linear congruential method to generate template random number. Our library can be easily adopted into their solution providing a more sophisticated random number generation.

Fig. 1. Compilation times of the quicksort of the boost::mpl::vector_c using the original boost::mpl::sort and the modified sort, where the pivot element is selected randomly, on sorted and non-sorted (random) data for various vector sizes. Measured with g++ 4.7.3 on Ubuntu x86.
Meredith L. Patterson mentions a “simple compile-time pseudo-random number generator” she implemented [16], but no further details are available.

6 Future works

Our goal was to provide the metaprogramming counterpart of the random number generator library of the STL. We ported the random number generator engines and all the random number distributions that generate integral values. However, we need to extend the library with distributions that generate floating point values e.g. normal distribution. As the language supports only integers in template arguments, we have to find a way to circumvent this limitation. Neither the standard library, nor third party libraries offer a ready solution, therefore we need to implement a floating point metatype first. Based on this metatype we can implement the remaining statistical distributions provided by the STL.

Our library is designed to be extensible. Further random number engines and distributions can be added. The engines provide a clean and simple interface so new engines and distributions can be created orthogonally.

7 Conclusion

Template metaprogramming plays essential role in library design in C++. Several language features and third party libraries supports that paradigm. However, due to the deterministic nature of template metapograms it was difficult to implement algorithms and data structures in undeterministic way. Since sometimes randomized algorithms and data structures are often less complex and
more efficient than their deterministic correspondents, it is important to generate (pseudo-)random numbers in a maintainable and effective way for template metaprograms.

We implemented random number engines that generate pseudorandom integer sequences with a uniform distribution and random number distributions that transform the generated pseudo-numbers into different statistical distributions. The library has a similar, but compile-time interface like the run-time random number generator of the STL to reduce the learning curve.

In this paper we presented our library and discussed its applicability with an example using boost::mpl. Besides, our library is designed to be extensible, thus one can easily add further engines and distributions to our library.

References