SF Bay Area ACCU
Dec 13, 2017

Systematic Generation of Data and Types in C++

Sumant Tambe
Sr. Software Engineer and Microsoft MVP
LinkedIn
@sutambe

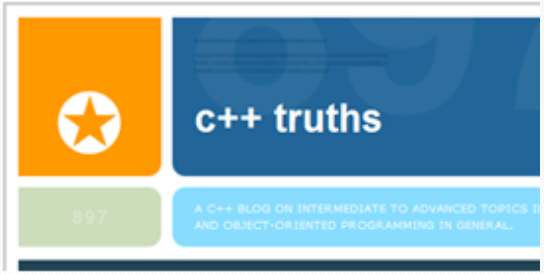
Alternative Title

Composable Generators and Property-based Testing in C++

Alternative Title

Functional Style API Design In Modern C++

Blogger

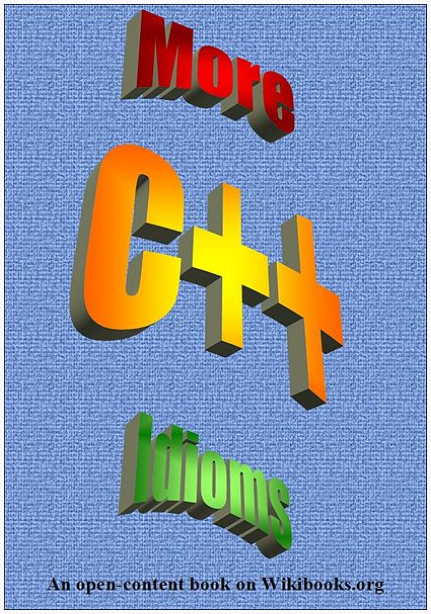


Coditation—Elegant Code for Big Data

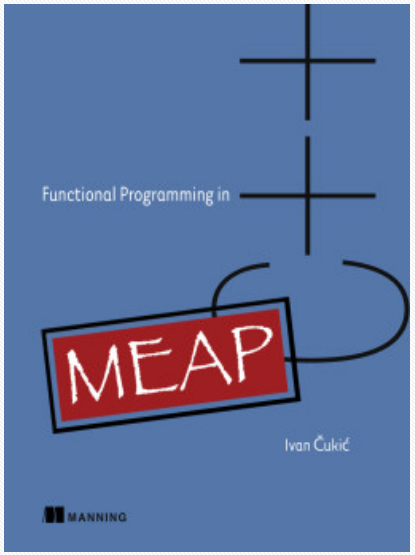
Open-source contributor

Apache kafka

Author



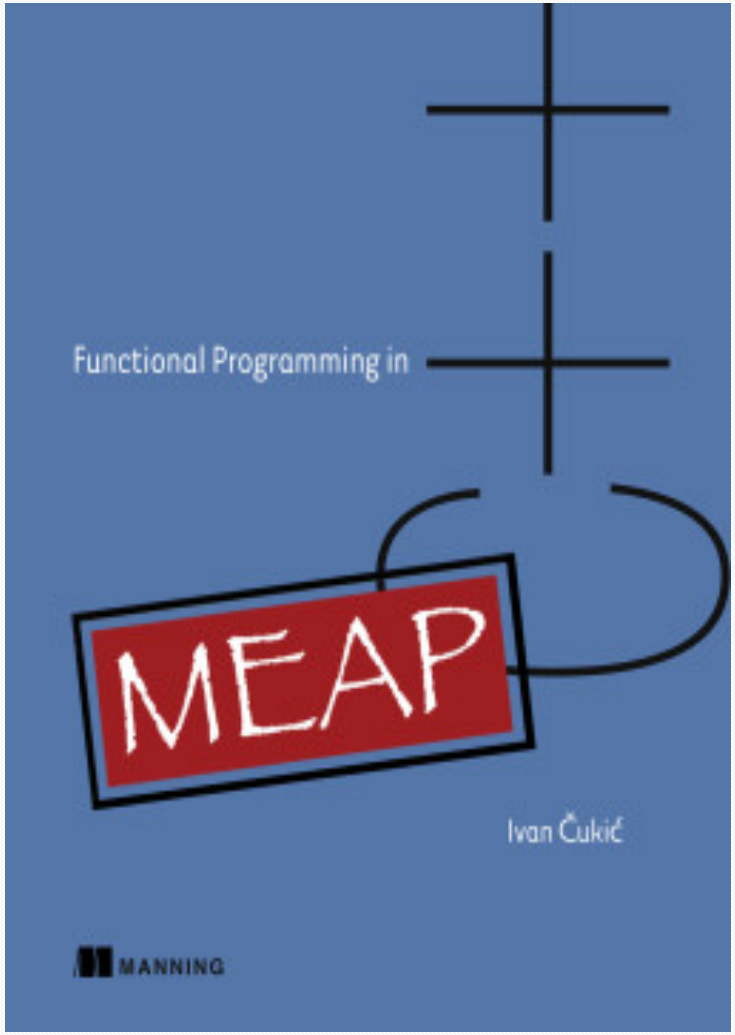
Reviewer



Since 2013 (Visual Studio and Dev Tech)

Functional Programming in C++ by Manning

Author
Ivan Cukic



This work was done in

Connectivity Software for the Industrial Internet of Things (IIoT)

Why should you care?

- If you write software for living and if you are serious about any of the following
  - Bug-free, Expressive code
  - Productivity
    - Modularity
    - Reusability
    - Extensibility
    - Maintainability
  - and having fun while doing all of the above!

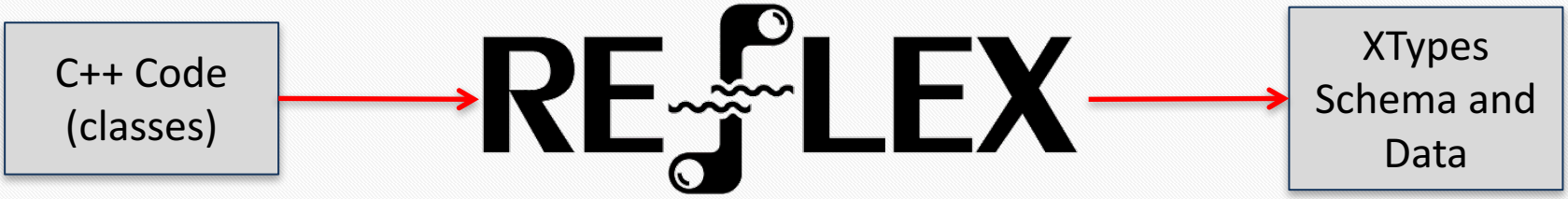
Agenda

- Property-based Testing (briefly)
- Why composable generators
- Implementation of a functional generator library
  - Less than 1500 lines of code
  - Wrapping a coroutine as a generator (if time permits)
- Mathematics of API design
  - Functor, Monoid, and Monad
- Type generators
  - Generators at compile-time

What's a Property

- Reversing a linked-list twice has no effect
  - `reverse(reverse(list)) == list`
- Compression can make things smaller
  - `compress(input).size <= input.size`
- Round-trip serialization/deserialization has no effect
  - `deserialize(serialize(input)) == input`

Testing RefleX



- RefleX is short for **Reflection for DDS-XTypes**
  - DDS-XTypes is like Avro, ProtoBuf, etc.
- Reflex is a library that acts like a code generator
- Serialization and Deserialization
  - Serializes **types** and **data**
  - Compile-time reflection on C++ structs/classes
- C++11/14
- Link

RefleX Example

```
class ShapeType
{
  std::string color_;
  int x_, y_, shapesize_;

public:
  ShapeType() {}
  ShapeType(const std::string & color,
            int x, int y, int shapesize)
    : color_(color), x_(x), y_(y),
      shapesize_(shapesize)
  {}

  std::string & color();
  int & x();
  int & y();
  int & shapesize();
};
```

```
#include "reflex.h"

REFLEX_ADAPT_STRUCT(
  ShapeType,
  (std::string, color(), KEY)
  (int,              x())
  (int,              y())
  (int,     shapesize()))
```

Uses boost Fusion (duh!)

A property-test in RefleX

```
template <class T>
void test_roundtrip_property(const T & in)
{
  T out;
  reflex::TypeManager<T> tm; // create Xtypes schema
  reflex::SafeDynamicData<T> safedd =
      tm.create_dynamicdata(in);  // create Xtypes data
  reflex::read_dynamicdata(out, safedd);

  assert(in == out); // must be true for all T
}
```

What is T?
What is `in`?

Property-based Testing

- Complementary to traditional example-based testing
- Specify post-conditions that must hold no matter what
- Encourages the programmer to think harder about the code
- More declarative
- Need data generators to produce inputs
- Need type generators to produce input types
- Free reproducers!
  - Good property-based testing frameworks shrink inputs to minimal automatically
- Famous
  - Haskell's QuickCheck
  - Scala's ScalaTest

Data Generators

Random Integer Generator

- The simplest data generator

```
size_t seed = time(nullptr);
srandom(seed);

size_t rand()
{
  return random();
}
```

- Alternatively

```
size_t seed =
  std::chrono::system_clock::now().time_since_epoch().count();

size_t rand()
{
  static std::mt19937 mt(seed);
  return mt();
}
```

Fundamental Data Generators

- A `bool` generator

```
bool bool_gen() {
  return rand() % 2;
}
```

- An uppercase character generator

```
char uppercase_gen() // A=65, Z=90
  return static_cast<char>('A' + rand() % 26))
}
```

Data Generators

- A range generator
  - Generates any number in the range (inclusive)

```
int range_gen(int lo, int hi)
  return lo + rand() % (hi - lo + 1);
}
```

- Inconvenient because args must be passed every time

Closure as generator

- A stateful range generator
  - Generates a random number in the given range

```
auto make_range_gen(int lo, int hi)
  return [lo, hi]() {
            return lo + rand() % (hi - lo + 1);
         }
}

auto r = range_gen(20, 25);
std::cout << r(); // one of 20, 21, 22, 23, 24, 25
```

Closure as generator

- The `oneof` generator

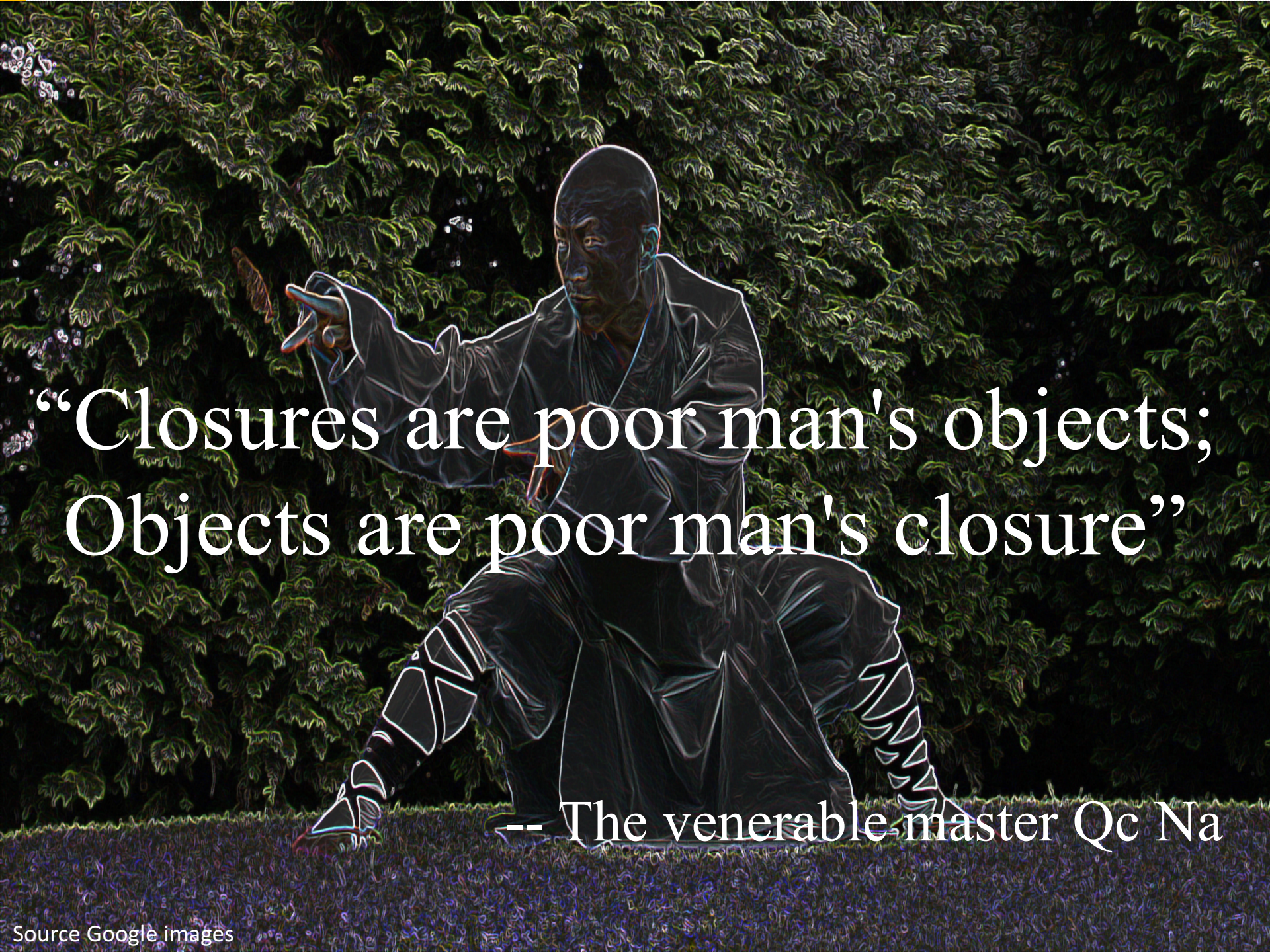
```
template <class T>
auto make_oneof_gen(std::initializer_list<T> list)
{
  return [options = std::vector<T>(list)]() {
    return *(options.begin() + rand() % options.size());
  };
}
auto days = { “Sun”, “Mon”, “Tue”, “Wed”, “Thu”, “Fri”, “Sat” }
auto gen = make_oneof_gen(days);

std::cout << gen(); // one of Sun, Mon, Tue, Wed, Thu, Fri, Sat
```

Closure vs Object

- Is function/closure a sufficient abstraction for <u>any</u> generator?

```
[]() {
    return blah;
}
```

- Answer depends upon your language
  - No. In C++



Source Google images

The Generator Class Template

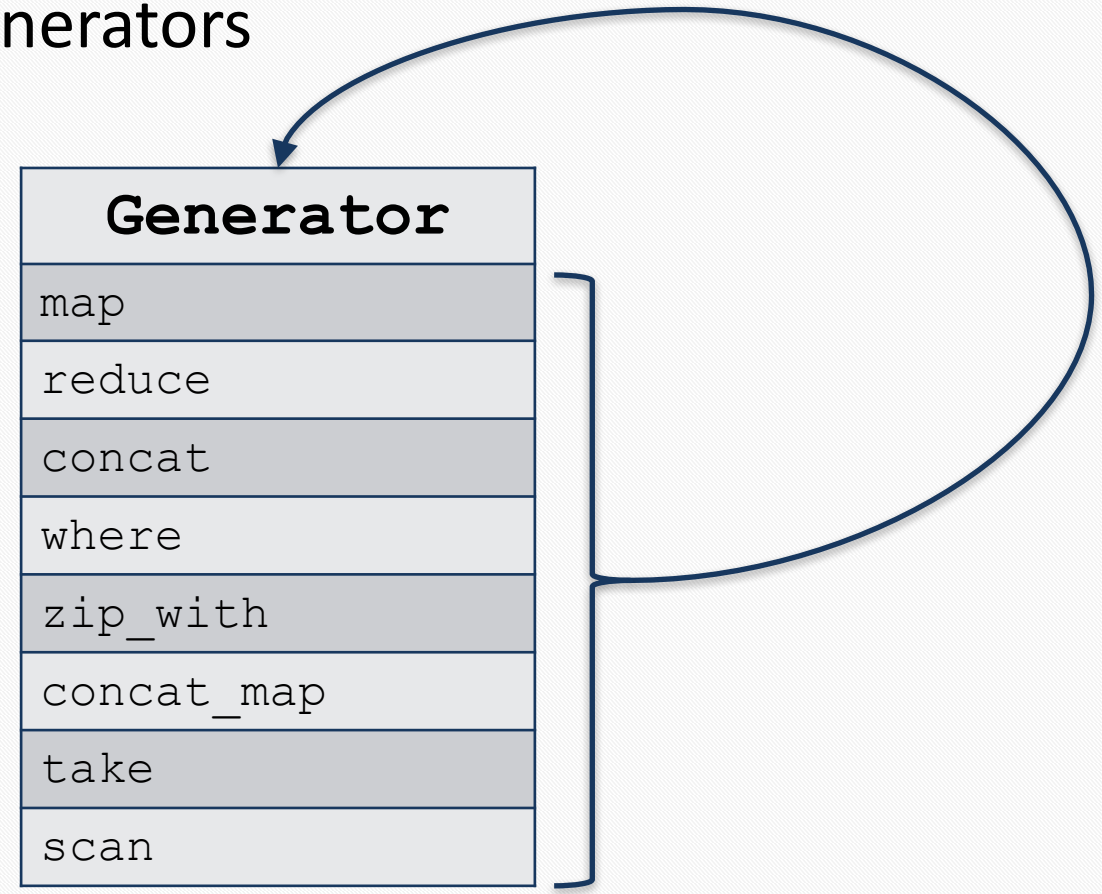
```
template <class T, class GenFunc>
struct Gen : private GenFunc
{
  using value_type = T;

  explicit Gen(GenFunc func)
    : GenFunc(std::move(func))
  { }

  T generate()
  {
    return GenFunc::operator()();
  }

  // ...
};
```

Creating Gen<T> from lambdas

```
template <class GenFunc>
auto make_gen_from(GenFunc&& func)
{
  return
    Gen<decltype(func()), GenFunc>(std::forward<GenFunc>(func));
}

template <class Integer>
auto make_range_gen(Integer lo, Integer hi)
{
  return make_gen_from([lo, hi]() {
     return static_cast<Integer>(lo + rand() % (hi - lo));
  });
}
```

The Real Power of Generators: Composition

- `make_gen_from` is low level
- Better—Producing complex generators from one or more simpler Generators



The Generator Class Template

```
template <class T, class GenFunc>
struct Gen : private GenFunc
{
  using value_type = T;

  explicit Gen(GenFunc func)
    : GenFunc(std::move(func))
  { }

  T generate()
  {
    return GenFunc::operator()();
  }

  auto map(...) const;
  auto zip_with(...) const;
  auto concat(...) const;
  auto concat_map(...) const;
  auto reduce(...) const;
  auto where(...) const;
  auto scan(...) const;
  auto take(...) const;
};
```

Mapping over a generator

```
vector<string> days = { "Sun", "Mon", "Tue", "Wed",
                        "Thu", "Fri", "Sat" };

auto range_gen  = gen::make_range_gen(0,6);
// random numbers in range 0..6

auto day_gen    = range_gen.map([&](int i) { return days[i]; });
// random days in range "Sun" to "Sat"

while(true)
  std::cout << day_gen.generate();
// random days in range "Sun" to "Sat"
```

f(int → string)

Gen<int> → Gen<string>

std::function<string(int)>

Generator is a

Functor.

You can **Map** over it.

Functor Laws

- Composition Law

gen.map(f).map(g) == gen.map(g ∘ f)

```
auto DAY_gen    = range_gen
                    .map([&](int i) { return days[i]; });
                    .map([](const string & str) {
                      return std::toupper(str, std::locale());
                    });
```

```
auto DAY_gen    = range_gen.map([&](int i) {
                     return std::to_upper(days[i], std::locale());
                  });
```

Functor Laws

- Identity Law

  gen.map(a -> a) == gen

```
auto gen2  == gen.map([](auto a) { return a; });
```

- Both gen and gen2 will generate the same data
  - Their identities are different but they are still the same

Implementing `map`

```
template <class T>
template <class Func>
auto Gen<T>::map(Func&& func) const &
{
  return make_gen_from(
           [self = *this, // copy
            func = std::forward<Func>(func)]() mutable {
               return func(self.generate());
         });
}
auto Gen<T>::map(Func&& func) &&
{
  return make_gen_from(
           [self = std::move(*this), // move
            func = std::forward<Func>(func)]() mutable {
               return func(self.generate());
         });
}
```

Generator is a

Monoid.

Monoid Laws

- Identity Law

  $$M \circ id = id \circ M = M$$

- Binary operation $\circ$ = append
- Appending to identity (either way) must yield the same generator

```
auto identity = gen::make_empty_gen<int>();

gen::make_inorder_gen({ 1, 2, 3}).append(identity)
==
identity.append(gen::make_inorder_gen({ 1, 2, 3});
```

Monoid Laws

- Associative Law

  $(f \circ g) \circ h = f \circ (g \circ h)$

- Binary operation $\circ$ = append
- Order of operation does not matter. (The order of operands does)
  - Not to be confused with commutative property where the order of operands does not matter

```
auto f = gen::make_inorder_gen({ 1, 2});
auto g = gen::make_inorder_gen({ 3, 4});
auto h = gen::make_inorder_gen({ 5, 6});

(f.append(g)).append(h) == f.append(g.append(h));
```

Creating Identity Generator
make_empty_gen

```
template <class T>
auto make_empty_gen()
{
  return make_gen_from([]() {
      throw std::out_of_range("empty generator!");
      return std::declval<T>();
    });
}
```

Gen::append

```
template <class UGen>
auto append(UGen&& ugen) const
{
  return make_gen_from(
   [tgen = *this,
    ugen = std::forward<UGen>(ugen),
    tdone = false,
    udone = false]() mutable {
    if(!tdone)
     {
       try {
         return tgen.generate();
       }
       catch(std::out_of_range &)
       {
         tdone = true;
       }
     }
    if(!udone)
    {
      try {
        return ugen.generate();
      }
      catch(std::out_of_range &) {
        udone = true;
      }
    }

    throw std::out_of_range(
    "concat: both generators done!");

  });

}
```

Zipping Generators

```
auto x_gen = gen::make_stepper_gen(0,100);
auto y_gen = gen::make_stepper_gen(0,200,2);

auto point_gen =
  x_gen.zip_with([](int x, int y)
                      return std::make_pair(x, y);
                 }, y_gen);

for(auto _:  { 1,2,3 })
  std::cout << point_gen.generate(); // {0,0}, {1,2}, {3,4}
end
```

`zip_with` is a generator of arbitrary struct/tuple values

Implementing `zip_with`

```
template <class T>
template <class Zipper, class... GenList>
auto Gen<T>::zip_with(Zipper&& func, GenList&&... genlist) const
{
  return make_gen_from(
    [self = *this,
     genlist...,
     func = std::forward<Zipper>(func)]() mutable {
          return func(self.generate(), genlist.generate()...);
    });
}
```

Generator is a

Monad.

Dependent Generators (concat_map)

```
1
12
123
1234
12345
1234
123
12
1
```

```
auto triangle_gen =
    gen::make_inorder_gen({1,2,3,4,5,4,3,2,1})
        .concat_map([](int i) {
            std::cout << "\n";
            return gen::make_stepper_gen(1, i);
        });

  try {
    while(true)
      std::cout << triangle_gen.generate();
  }
  catch(std::out_of_range &) {
  }
```

Monad Laws

```
auto f = [](int i) { return gen::make_stepper_gen(1, i); };
auto g = [](int i) { return gen::make_stepper_gen(i, 1, -1); };
auto M = gen::make_single_gen(300);

// left identity
M.concat_map(f) == f(300);

// right identity
M == M.concat_map([](int i) {
          return gen::make_single_gen(i);
       });
// associativity
M.concat_map(f).concat_map(g)
==
M.concat_map([f,g](int i) mutable {
   return f(i).concat_map(g);
}));
```

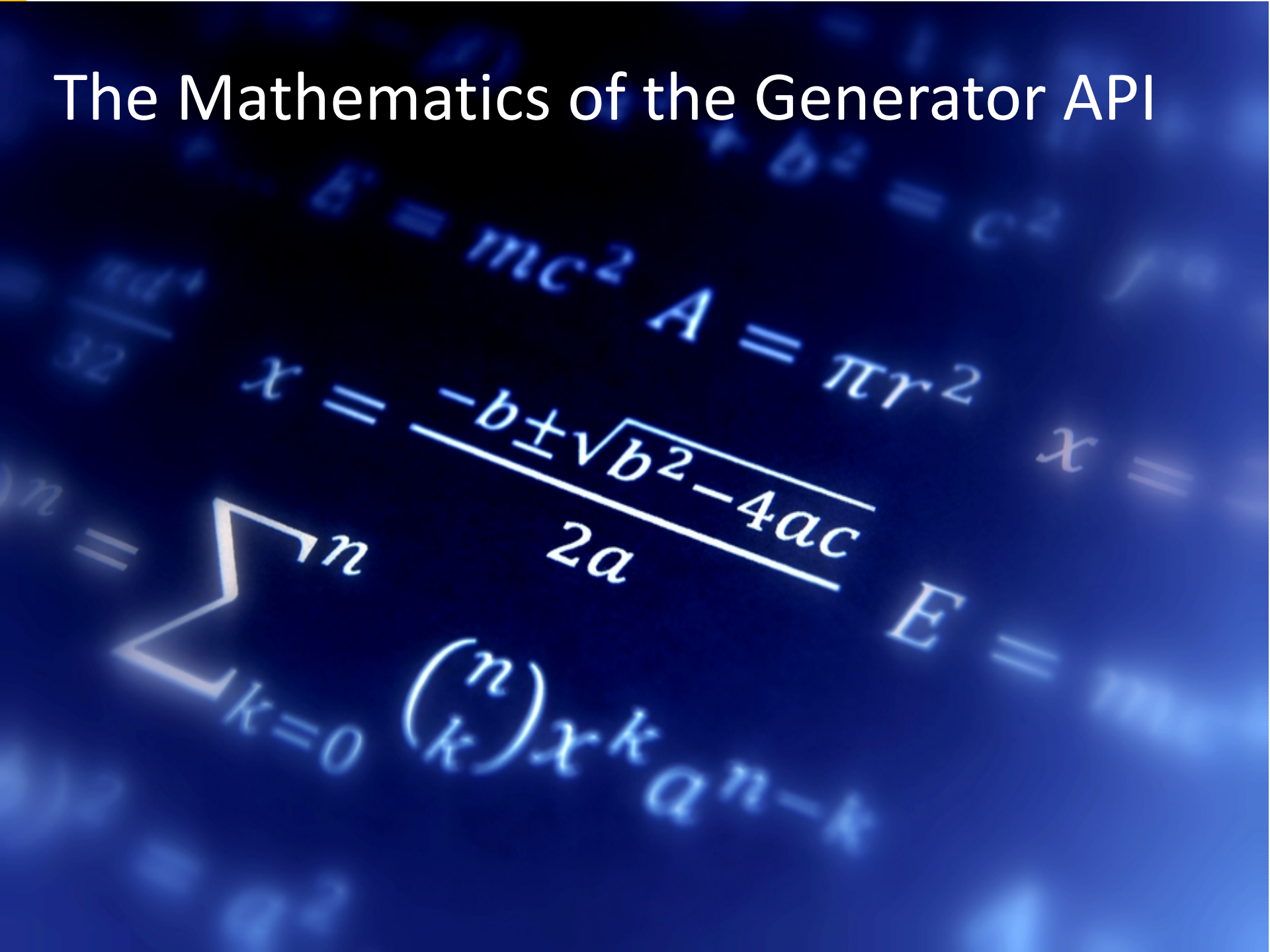
Monad's "return" is make_single_gen

```
template <class T>
auto make_single_gen(T&& t)
{
  return make_gen_from([t = std::forward<T>(t),
                        done = false]() mutable {
      if(!done)
      {
        done = true;
        return t;
      }
      else
        throw std::out_of_range("single generator completed!");
    });
}
```

Monad's "bind" is `Gen::concat_map`

- See generator.h

The Mathematics of the Generator API



Generators

- General design principal
  - Create a language to solve a problem
  - API is the language
  - Reuse parsing/compilation of the host language (duh!)
- The “language” of Generators
  - Primitive values are basic generators
  - Complex values are composite generators
  - All APIs result in some generator
    - `map`, `zip_with`, `concat_map`, `reduce`, etc.
  - I.e., generators form an **algebra**

Algebra

A first-class, lawful abstraction

+

Compositional* API

*Composition based on mathematical concepts

What Math Concepts?

- Algebraic Structures from Category Theory
  - Monoid
  - Functor
  - Monad
  - Applicative

- Where to begin?
  - Haskell
  - But innocent should start with Scala perhaps

Back to the property-test in RefleX

```
template <class T>
void test_roundtrip_property(const T & in)
{
  T out;
  reflex::TypeManager<T> tm;
  reflex::SafeDynamicData<T> safedd =
      tm.create_dynamicdata(in);
  reflex::read_dynamicdata(out, safedd);

  assert(in == out);
}
```

What is T?
What is `in`?

we can generate

Random Types at Compile-time

if we can generate

Random Numbers at Compile-time

Random Numbers at Compile-time

```c
#include <stdio.h>

enum test { Foo = RANDOM };

int main(void)
{
  enum test foo = Foo;
  printf("foo = %d\n", foo);
  return 0;
}
```

```
$ gcc -DRANDOM=$RANDOM main.c -o main
$ ./main
foo = 17695
```

Random Numbers at Compile-time

```
constexpr int LFSR_bit(int prev) {
   return (((prev >> 0) ^ (prev >> 2) ^
            (prev >> 3) ^ (prev >> 5)) & 1);
}

constexpr int LFSR(int prev) {
  return ((prev >> 1) | (LFSR_bit(prev) << 15));
}


enum test { Foo = RANDOM,  Bar = LFSR(Foo) };
int main(void) {
  enum test foo = Foo;
  enum test bar = Bar;
  printf("foo = %d, bar = %d\n", foo, bar);
}
```

```
$ gcc -DRANDOM=$RANDOM main.c -o main
$ ./main
foo = 17695, bar = 41615
```

Linear Feedback Shift Register (LFSR)

- Simple random number generator
- 4 bit Fibonacci LFSR
- Feedback polynomial = $X^4 + X^3 + 1$
- Taps 4 and 3

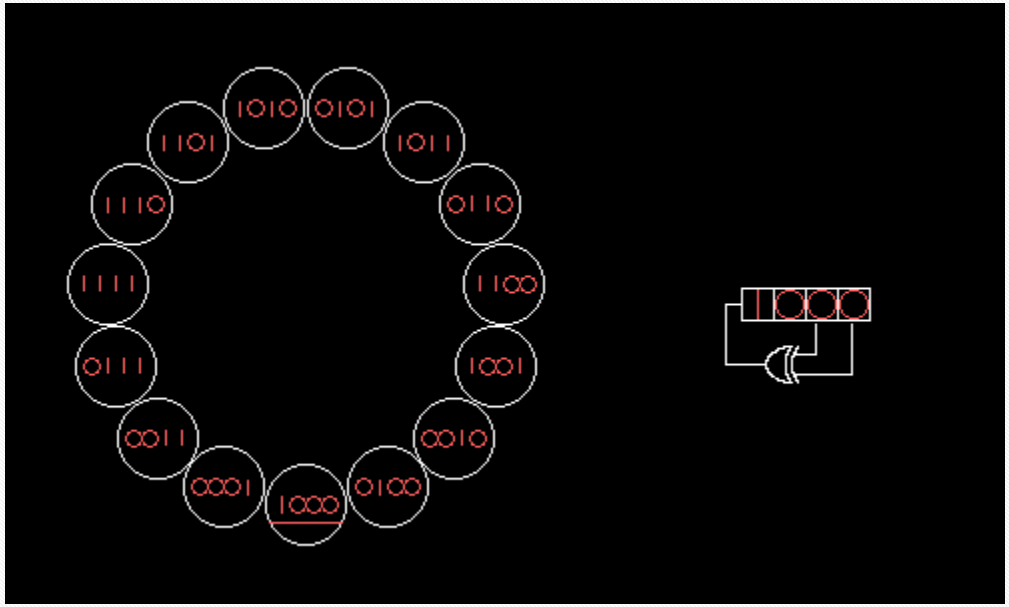


Image source: wikipedia

Linear Feedback Shift Register (LFSR)

- A 16 bit feedback polynomial
  - $X^{16} + X^{14} + X^{13} + X^{11} + 1$
- Taps: 16, 14, 13, and 11

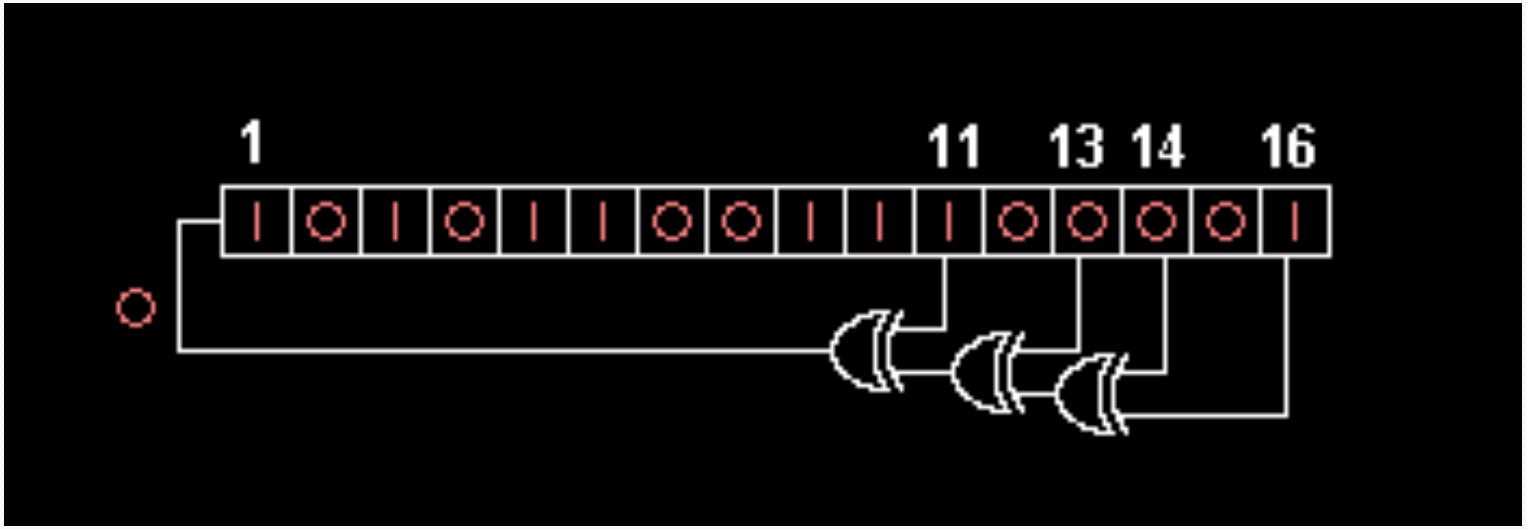


Image source: wikipedia

Random Type Generation at Compile-Time

```
template <int I> struct TypeMap;

template <> struct TypeMap<0> {
    typedef int type;
};

template <> struct TypeMap<1> {
    typedef char type;
};

template <> struct TypeMap<2> {
    typedef float type;
};

template <> struct TypeMap<3> {
    typedef double type;
};
```

Random Type Generation at Compile-Time

```
#include <boost/core/demangle.hpp>

template <int seed>
struct RandomTuple {
  typedef typename TypeMap<LFSR(seed) % 4>::type First;
  typedef typename TypeMap<LFSR(LFSR(seed)) % 4>::type Second;
  typedef typename TypeMap<LFSR(LFSR(LFSR(seed))) % 4>::type Third;

  typedef std::tuple<First, Second, Third> type;
};
int main(void) {
  auto tuple = RandomTuple<RANDOM>::type();
  printf("tuple = %s",
    boost::core::demangle(typeid(tuple).name()).c_str());
}
```

```
$ g++ -DRANDOM=$RANDOM main.cpp -o main
$ ./main
foo = 11660, bar = 5830
tuple = std::tuple<float, double, char>
```

Live code

Tuple Value Generator (zip)

```
template <class... Args>
struct GenFactory<std::tuple<Args...>>
{
  static auto make()
  {
    return make_zip_gen([](auto&&... args) {
        return std::make_tuple(std::forward<decltype(args)>(args)...);
      },
      GenFactory<Args>::make()...);
  }
};

int main(void) {
  auto tuple_gen =
     GenFactory<RandomTuple<RANDOM>::type>::make();
  auto tuple = tuple_gen.generate();
}
```

Tuples all the way down…

- Nested type generation using recursive `TypeMap`
- Classic template meta-programming
- See type_generator.h

Type Generators Demystified

- Random seed as command line preprocess argument
- Compile-time random numbers using LFSR `constexpr`
- C++ meta-programming for type synthesis
- `std::string` generator for type names and member names
- RefleX to map types to typecode and dynamic data

- Source: https://github.com/sutambe/cpp-generators

RefleX Property-test Type Generator

Github: https://github.com/rticommunity/rticonnextdds-reflex/blob/develop/test/generator

- "Good" Numbers
  - 31415 (3 nested structs)
  - 32415 (51 nested structs)
  - 2626 (12 nested structs)
  - 10295 (15 nested structs)
  - 21525 (41 nested structs)
  - 7937 (5 nested structs)
  - …
- "Bad" Numbers
  - 2152 (test seg-faults) `sizeof(tuple)` > 2750 KB!

Thank You!