Lab Class Scientific Computing 2022, WISM454

Adriaan Graas, Week 7

LCRNG

Implementations

Exercise 1.1

```
struct LCRNG {
    unsigned a, c, m;
};


// `next` defined for `struct LCRNG`
... next(struct LCRNG * rng, ...);


// `UniformDistribution` too
struct UniformDistribution{
   struct LCNRG * rng;
   ...
};


// `draw` depending on `Uniform distribution`
... draw(struct UniformDistribution...);
```

Different *variations* of LCRNGs could have different types

```
struct SchrageLCRNG {
    unsigned a, m;
};
```

For example **Park-Miller**. An LCRNG that would overflow and needed special computations. We assumed *c* was always zero.

```
struct FastModuloLCRNG {
    unsigned a, c;
    unsigned power_two;
};
```

Like **Randu**. An LCRNG that had a power of two as *m* and could be computed in a fast way.

```
struct TruncatingLCRNG {
    unsigned a, c, m;
    unsigned shift;
};
```

For example **SUN**. An LCRNG that needed the truncation of the right-most bits, because it was not very random.

However, different types would result in code duplication

```
struct SchrageLCRNG {
    unsigned a, m;

};

... next_schrage(...);

struct SchrageUniformDistribution{
   struct SchrageLCNRG * rng;
   ...
};

... draw_schrage(...);
... efficiency_schrage(...);
...
```

```
struct FastModuloLCRNG {
    unsigned a, c;
    unsigned power_two;
};

... next_fastmodulo(...);

struct FastModuloUniformDistribution{
   struct FastModuloLCNRG * rng;
   ...
};

... draw_fastmodulo(...);
... efficiency_fastmodulo(...);
...
```

```
struct TruncatingLCRNG {
    unsigned a, c, m;
    unsigned shift;
};

... next_truncating(...);

struct TruncatingUniformDistribution{
   struct TruncatingLCNRG * rng;
   ...
};

... draw_truncating(...);
... efficiency_truncating(...);
...
```

So, having multiple types was (agreeably) not a great idea

1. Everybody settled on having a single **struct LCRNG.**
2. But to handle the variations, we had to be inventive:
    - **Set m=0 for the QUICK generator;**
    - **Find a way to recognize when to use Schrage’s trick;**
    - **Find a way when to use the >> shift in SUN generator;**
    - **Find a way to truncate numbers when m was a power of two.**

Avoiding code duplication, inspired by your projects, #1: macro’s!

```
#define SCHRAGE

struct LCRNG { unsigned a, c, m; };

unsigned next(struct LCRNG * rng, unsigned x) {
#ifdef SCHRAGE
    // perform Schrage's method
#else
    return rng->a * rng->c % rng->m;
#endif
}
```

- Idea: just redefine the *next* function for (the) different type. Choose the implementation based-off a preprocessor macro.

Utrecht University

Avoiding code duplication, inspired by your projects, #1: macro’s!

```c
#define SCHRAGE

struct LCRNG { unsigned a, c, m; };

unsigned next(struct LCRNG * rng, unsigned x) {
#ifdef SCHRAGE
    // perform Schrage's method
#else
    return rng->a * rng->c % rng->m;
#endif
}
```

- Idea: just redefine the *next* function for (the) different type. Choose the implementation based-off a preprocessor macro.
- Advantages:
    - **Compilation to potentially fastest code.**
- Disadvantages:
    - **Allows only one option to be used in any program. Not very flexible.**
    - **Requires recompilation when changing the option.**

Avoiding code duplication, inspired by your projects, #2: the big next function

```c
struct LCRNG { unsigned a, c, m; };

unsigned next(struct LCRNG * rng, unsigned x) {
    if (rng->c == 0) {
        // mixed generator, probably something Schrage
    } else if (rng->m == 0) {
        // that must be Quick
    } else if (is_power_of_two(rng->m)) {
        // something Randu-like
    } else {
        return (rng->a * x + rng->c) % rng->m;
    }
}
```

- Idea: one big *next* function to solve it all.

Avoiding code duplication, inspired by your projects, #2: the big next function

```
struct LCRNG { unsigned a, c, m; };

unsigned next(struct LCRNG * rng, unsigned x) {
    if (rng->c == 0) {
        // mixed generator, probably something Schrage
    } else if (rng->m == 0) {
        // that must be Quick
    } else if (is_power_of_two(rng->m)) {
        // something Randu-like
    } else {
        return (rng->a * x + rng->c) % rng->m;
    }
}
```

- Idea: one big *next* function to solve it all.
- Advantages:
  - **Keeps the LCRNG type clean.**
  - **Easy reusing code between different types.**
- Disadvantages:
  - **Unfortunately, performing all the if-statements in the *next* is slow.**
  - **Cannot safely deduce when to use Schrage's or a >> (SUN-style) generator.**

Utrecht University

Avoiding code duplication, inspired by your projects, #2B: deferred next functions

```
struct LCRNG { unsigned a, c, m; };

unsigned schrage_next(struct LCRNG * rng, unsigned x);
unsigned quick_next(struct LCRNG * rng, unsigned x);
unsigned mod32_next(struct LCRNG * rng, unsigned x);

unsigned next(struct LCRNG * rng, unsigned x) {
    if (rng->c == 0) {
        schrage_next(rng, x);
    } else if (rng->m == 0) {
        quick_next(rng, x);
    } else if (is_power_of_two(rng->m)) {
        mod32_next(rng, x);
    } else {
        return (rng->a * x + rng->c) % rng->m;
    }
}
```

- Idea: same as before. One big *next* function to solve it all. Now deferring the computation to sub functions.
- A bit more organized, but fundamentally the same as idea #2.

Avoiding code duplication, inspired by your projects, #3: complex type LCRNG

```c
struct LCRNG {
    unsigned a, c, m;
    int uses_schrage, sun_shift, power_two;
};

unsigned next(struct LCRNG * rng, unsigned x) {
    if (rng->use_schrage == 1) {
        // do Schrage
    } else if (rng->m == 0) {
        // that must be Quick
    } else if (rng->truncate_bits > 0) {
        // something Randu-like
    } else {
        x = rng->a * x + rng->c) % rng->m;
    }
}
```

- Idea: one big *next* to solve it all. Storing extra parameters in the struct.

Avoiding code duplication, inspired by your projects, #3: complex type LCRNG

```
struct LCRNG {
    unsigned a, c, m;
    int uses_schrage, sun_shift, power_two;
};

unsigned next(struct LCRNG * rng, unsigned x) {
    if (rng->use_schrage == 1) {
        // do Schrage
    } else if (rng->m == 0) {
        // that must be Quick
    } else if (rng->truncate_bits > 0) {
        // something Randu-like
    } else {
        x = rng->a * x + rng->c) % rng->m;
    }
}
```

- Idea: one big *next* to solve it all. Storing extra parameters in the struct.
- Advantages:
  - **Fast if-statements in the *next*.**
  - **Can make the *next* work for combinations of different LCRNG types.**
- Disadvantages:
  - **LCRNG type is overloaded with members that do not make sense for all LCRNGs.**
    - E.g., ‘uses_schrage’ and ‘truncate_bits’ may be incompatible.

Avoiding code duplication, inspired by your projects, #3B: labeled LCRNGs

```
struct LCRNG {
    unsigned a, c, m;
    int label;
};

unsigned next(struct LCRNG * rng, unsigned x) {
    if (rng->label == 1) {
        // Schrage
    } else if (rng->label == 2) {
        // Quick
    } else if (rng->label == 3) {
        // Randu
    } else {
        x = rng->a * x + rng->c) % rng->m;
    }
}
```

- Idea: one big *next* to solve it all. Storing a label in the struct.
- Same as #3: now a label is a number that corresponds to a combination of whether or not to use Schrage, a shift for SUN, and/or a power of two.
- A bit cleaner, a bit less flexible.
- Also possible: an *enum* type to give names to the labels.

Utrecht University

Avoiding code duplication, inspired by your projects, #4: pointer types

```
struct LCRNG {
    unsigned a, c, m;
    unsigned (*next_func)(struct LCRNG *, unsigned);
};

unsigned next_schrage(struct LCRNG *rng, unsigned x) {
    ... }

unsigned next_quick(struct LCRNG *rng, unsigned x) {
    ... }

unsigned next_default(struct LCRNG *rng, unsigned x) {
    return (rng->a * x + rng->c) % rng->m;
}
```

- Idea: multiple *next* functions. Store function pointer to be used in the LCRNG struct.

Avoiding code duplication, inspired by your projects, #4: pointer types

```
struct LCRNG {
    unsigned a, c, m;
    unsigned (*next_func)(struct LCRNG *, unsigned);
};

unsigned next_schrage(struct LCRNG *rng, unsigned x) {
    ... }

unsigned next_quick(struct LCRNG *rng, unsigned x) {
    ... }

unsigned next_default(struct LCRNG *rng, unsigned x) {
    return (rng->a * x + rng->c) % rng->m;
}
```

- Idea: multiple *next* functions. Store function pointer to be used in the LCRNG struct.

- Advantages:
    - **After macro's, the fastest approach, as no if-statements are needed.**
    - **Clean organization of *next*.**
- Disadvantages:
    - **Would need a new *next* for each thinkable combination of generators.**
    - **Can handle custom next functions, for specific >> choices or powers of two, but is complicated.**

Object-oriented programming

(Subtype) polymorphism

What is polymorphism?

- **Subtype polymorphism** is building a single *interface* to work for a variation of types.
    - **In C++, the term *interface* does not have a technical meaning**
    - **Can be just a superclass, doesn't necessarily have to be *abstract***

- Using *inheritance*, "subtyping", different implementations can be used to *implement* the interface.

- Two other forms of *polymorphism*:
    - ***Ad hoc* polymorphism: function overloading**
        - For example constructor overloading
    - ***Parametric* polymorphism: template types (later lecture)**

Polymorphism example

```
class Animal { // interface
public:
    virtual string sound() = 0;
};

class Cat : public Animal {
public:
    string sound() override { return "meow"; };
};

class Dog : public Animal {
public:
    string sound() override { return "wooff"; }
};
```

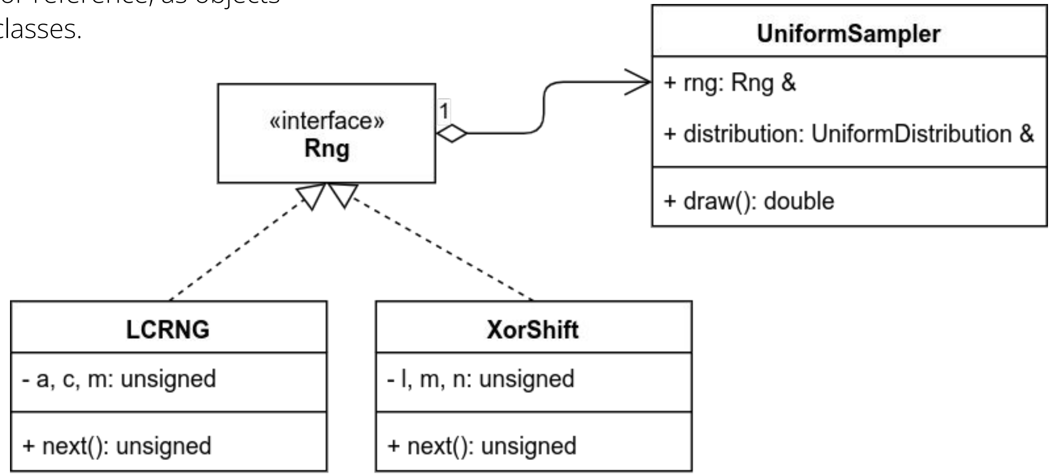
Polymorphism example

```
class Animal { // interface
public:
    virtual string sound() = 0;
};

class Cat : public Animal {
public:
    string sound() override { return "meow"; };
};

class Dog : public Animal {
public:
    string sound() override { return "wooff"; }
};
```

```
void explanation(Animal & animal) {
    cout << "This animal says: "
         << animal.sound() << "!" << endl;
}

int main() {
    Cat kitty {};
    Dog doggy {};

    explanation(kitty); // This ... meow!
    explanation(doggy); // This ... wooff!

    return 0;
}
```

Polymorphism of the Rng class

- We have an interface ***Rng*** and ***LCRNG*** and ***XorShift*** as subtypes.
- **UniformSampler** depends on **Rng** (and not on one of the implementations).
- **Rng** must be stored as a pointer or reference, as objects cannot be created from abstract classes.



Object-oriented programming

Dynamic dispatch

What is dynamic dispatch?

- Whenever a member function is marked *virtual*, C++ will use *dynamic dispatch* to call the function:
  - **The right implementation of the function will be selected at run-time: serious overhead.**
  - **In *static dispatch* the compiler will select the implementation at compile time.**

- Advantage: effective type of an object can be changed during the program's run.
- Disadvantage: dispatch mechanism involves additional computational cost.

```
void explanation(Animal & animal) {
    cout << "This animal says: "
         << animal.sound() << "!" << endl;
}
```

Classes are implemented as C-style data types and functions

```
class Dog {
public:
    virtual string sound(...) { ... };
    double weight;
};
```

```
class Dog {
    double weight;
};
```

→ Objects are like struct data types in memory.

```
void sound(Dog *this,
           ...) {
    ...
};
```

→ Member functions are C-style functions in memory. The object is passed through a special first argument.

C++ dynamic dispatch: vtables

- The compiler puts a small table, called a **vtable**, into memory for each class (parent or child).
    - **The vtable contains the virtual methods of the class.**
- Each *object* will have a special pointer, a **vpointer**, in memory, referring to a vtable.
- A function call uses the vpointer to find out where to find the implementing method is in memory.
    - **vtables of derived types have similar layout to speed-up the lookup process.**

```
class Dog {
public:
    virtual string sound() { ... "woof"; };
    double weight;
};

class PitBull : public Dog {
public:
    string sound() override { ... "WOOF"; };
};

...
Dog small_doggy {3.0};

PitBull big_dog {10.0};
```

vtable Dog
- function pointer to Dog::sound()

vtable PitBull
- function pointer to PitBull::sound()

memory for “small_doggy”
- vtable pointer to Dog
- weight: 3.0

memory for “big_dog”
- vtable pointer to PitBull
- weight: 10.0

Utrecht University

This week

- Today / this week:
    - **Exercise 2.5: Rejection method**
        - For rejection with uniform upper bound, there are multiple solutions.
    - **Exercise 2.6: Distribution arithmetic**
        - To prevent code duplication in Ex. 2.6.4 requires *multiple inheritance*, and may be challenging. If, however, you'd like to give it a go there is Ex. 2.6.5.
- Remark: adding appropriate plots/experiments to the report is up to you.