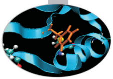
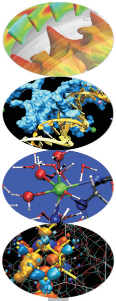


Scientific and Technical Computing in C++

Part 2 A C with Class

Luca Ferraro Mario Tacconi

CINECA Roma - SCAI Department



Rome, November 2016

Outline

1. Do you Need an Object?
    - Random Number Generators
    - A Classy Solution
    - Classes at Work
    - More Touches of Class
    - Polishing it Up
    - Wrapping it Up
2. Inheritance and Polymorphism
3. Class I/O

Lagged Fibonacci RNGs

- Let's imagine we have a simple-minded implementation of a pretty good RNG
- Defined by the recurrence relation:
  $x_i = (x_{i-l} + x_{i-k}) \mod 2^M$
- For specific, known (k, l) pairs the sequence has a period of $(2^k - 1)2^{M-1}$ terms
- Not necessarily the best RNG, but good enough for our purposes

- We want to make it better:
  1. allow for many independent generators in a program
  2. give users control on length (i.e. occupied memory, i.e. k)
  3. hide implementation details (i.e. avoiding users 'accidentally' fiddling with internals)

Simple Minded Lagged Fibonacci RNG

```
// Lagged Fibonacci RNG
// Possible (l, k) pairs could be, among others: (24, 55), (31, 73), (27,98)
// See Knuth, The Art of Computer Programming, v. 2, p. 26ff

#include <stdlib.h>
#include "lfrng.h"

#define LFRNG_K 55
#define LFRNG_L 24

static unsigned lfhstr[LFRNG_K];
static unsigned lfimk;
static unsigned lfiml;

void lfrng_init() {
    int i;

    for(i=0; i<LFRNG_K; ++i)
        lfhstr[LFRNG_K-i-1] = rand();

    lfimk = LFRNG_K-1;
    lfiml = LFRNG_L-1;
}
unsigned lfrng_draw() {
    unsigned r;

    r = lfhstr[lfimk] + lfhstr[lfiml];
    lfhstr[lfimk] = r;
    if (lfimk-- == 0) lfimk = LFRNG_K-1;
    if (lfiml-- == 0) lfiml = LFRNG_K-1;
    return r;
}
```

A C Solution: `lfrng.h`

- Let’s define an opaque type, without publishing its internals
- Let’s restrict its manipulation to functions in a sober API

- Users will only access what’s published in the `lfrng.h` header:

```
#ifndef LFRNG
#define LFRNG

struct LFRNG_inn;

typedef struct LFRNG_inn *LFrng;

LFrng lfrng_create(unsigned n);
void lfrng_init(LFrng g);
unsigned lfrng_draw(LFrng g);
void lfrng_destroy(LFrng g);
#endif
```

A C Solution: `lfrng.c` part 1 of 3

```
// Multiple Lagged Fibonacci RNGs
// Possible (l, k) pairs could be, among others: (24, 55), (31, 73), (27,98)
// See Knuth, The Art of Computer Programming, v. 2, p. 26ff

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include "lfrng.h"

#define LFRNGL_K 98
#define LFRNGL_L 27
#define LFRNGM_K 73
#define LFRNGM_L 31
#define LFRNGS_K 55
#define LFRNGS_L 24

struct LFRNG_inn {
    unsigned k, l;
    unsigned imk, iml;
    unsigned *hstr;
};
```

continues on next slide...

A C Solution: `lfrng.c` part 2 of 3

```
LFrng lfrng_create(unsigned n) {
    LFrng g;

    g = calloc(1, sizeof(*g));
    if (!g) {
        fprintf(stderr, "Not enough memory!\n");
        exit(-2);
    }

    g->k = LFRNGL_K;
    g->l = LFRNGL_L;
    if (n <= LFRNGS_K) {
        g->k = LFRNGS_K;
        g->l = LFRNGS_L;
    } else if (n <= LFRNGM_K) {
        g->k = LFRNGM_K;
        g->l = LFRNGM_L;
    } else if (n > LFRNGL_K)
        errno = EDOM;

    g->hstr = calloc(g->k, sizeof(unsigned));

    if (!g->hstr) {
        fprintf(stderr, "Not enough memory!\n");
        exit(-2);
    }

    return g;
}
```

continues on next slide...

A C Solution: `lfrng.c` part 3 of 3

```
void lfrng_destroy(LFrng g) {
    free(g->hstr);
    free(g);
}


void lfrng_init(LFrng g) {
    int i;

    for(i=0; i<g->k; ++i)
        g->hstr[g->k-i-1] = rand();

    g->imk = g->k-1;
    g->iml = g->l-1;
}


unsigned lfrng_draw(LFrng g) {
    unsigned r;

    r = g->hstr[g->imk] + g->hstr[g->iml];
    g->hstr[g->imk] = r;
    if (g->imk-- == 0) g->imk = g->k-1;
    if (g->iml-- == 0) g->iml = g->k-1;

    return r;
}
```

Making It More Robust

- User guide:
  1. create a `LFrng` using `lfrng_create()`
  2. initialize it using `lfrng_init()`
  3. call `lfrng_draw()` on it, from 1 to $(2^{\mathtt{k}} - 1)2^{31} - \mathtt{k}$ times
  4. destroy it using `lfrng_destroy()`

- Wait! What if step 2 is forgotten?
  - a sequence of one term: 0
  - separate initialization makes little sense

- Let's fix it

A C Solution: `lfrng.c` part 2 of 3 Revised

```
LFrng lfrng_create(unsigned n) {
    LFrng g;

    g = calloc(1, sizeof(*g));
    if (!g) {
        fprintf(stderr, "Not enough memory!\n");
        exit(-2);
    }

    g->k = LFRNGL_K;
    g->l = LFRNGL_L;
    if (n <= LFRNGS_K) {
        g->k = LFRNGS_K;
        g->l = LFRNGS_L;
    } else if (n <= LFRNGM_K) {
        g->k = LFRNGM_K;
        g->l = LFRNGM_L;
    } else if (n > LFRNGL_K)
        errno = EDOM;

    g->hstr = calloc(g->k, sizeof(unsigned));

    if (!g->hstr) {
        fprintf(stderr, "Not enough memory!\n");
        exit(-2);
    }

    lfrng_init(g);

    return g;
}
```

Adding Functionalities

- In need of a floating point RNG? Just include **`limits.h`** and add:

```
double lfrng_frand(LFrng g) {
    return lfrng_draw(g)/(double)UINT_MAX;
}
```

- Busy with heads and tails? Include **`bool.h`** too and add:

```
bool lfrng_toss(LFrng g) {
    return lfrng_draw(g) > (UINT_MAX/2);
}
```

- And so on...

Still Dissatisfying

- OK, init is automated, but what if creation is forgotten?
  - A segmentation fault, if we are lucky
- And what if the call to `lfrng_destroy()` is ‘omitted’?
  - A memory leak, if the program does it in a cycle
- And what if an array of RNGs is needed?
  - Each one must be created and destroyed explicitly
- `lfrng_draw()`, `lfrng_frand()`, `lfrng_toss()`: what if the wrong one is called?
  - A very surprising bug!

lfrng.h

Objects
RNGs
Class
Using Classes
More Class
Polishing
Wrap Up

Inheritance
Coins
FP RNGs
Heritage

Class I/O
Basics
Inheriting I/O

```
// Multiple Lagged Fibonacci RNGs
// See Knuth, The Art of Computer Programming, v. 2, p. 26ff
#ifndef LFRNG_H
#define LFRNG_H

namespace LFRNG {

class rng {
    unsigned k, l;
    unsigned imk, iml;
    unsigned *hstr;

    const static unsigned l_k = 98;
    const static unsigned l_l = 27;
    const static unsigned m_k = 73;
    const static unsigned m_l = 31;
    const static unsigned s_k = 55;
    const static unsigned s_l = 24;

public:
    rng(unsigned n);
    ~rng();
    void init();
    unsigned draw();
};

} //namespace LFRNG

#endif
```

Enter `class`

- `class` defines a data type that ties together:
    - data members
    - function members (a.k.a. methods)
- By default, class members are private
    - I.e. only accessible in the class scope
    - `public` members must be explicitly tagged as such
    - `private` members may also be tagged explicitly, if you like
    - C++ `struct`s are actually the same, only the default accessibility differs (default to public accessibility)
- Data members can be `const static`:
    - as usual, `const` means it cannot be writen to
    - `static` means there is one and only one instance of the member, common to all instances of the class
    - it's the preferred way of defining class specific constants without polluting other scopes

lfrng.h

```
// Multiple Lagged Fibonacci RNGs
// See Knuth, The Art of Computer Programming, v. 2, p. 26ff
#ifndef LFRNG_H
#define LFRNG_H

namespace LFRNG {

class rng {
private:
    unsigned k, l;
    unsigned imk, iml;
    unsigned *hstr;

    const static unsigned l_k = 98;
    const static unsigned l_l = 27;
    const static unsigned m_k = 73;
    const static unsigned m_l = 31;
    const static unsigned s_k = 55;
    const static unsigned s_l = 24;

public:
    rng(unsigned n);
    ~rng();
    void init();
    unsigned draw();
};

} //namespace LFRNG

#endif
```

lfrng.h: struct Equivalence

```
// Multiple Lagged Fibonacci RNGs
// See Knuth, The Art of Computer Programming, v. 2, p. 26ff
#ifndef LFRNG_H
#define LFRNG_H

namespace LFRNG {

struct rng {
    rng(unsigned n);
    ~rng();
    void init();
    unsigned draw();

private:
    unsigned k, l;
    unsigned imk, iml;
    unsigned *hstr;

    const static unsigned l_k = 98;
    const static unsigned l_l = 27;
    const static unsigned m_k = 73;
    const static unsigned m_l = 31;
    const static unsigned s_k = 55;
    const static unsigned s_l = 24;
};

} //namespace LFRNG

#endif
```

Methods

- Must be declared inside the class declaration
- Can access all members of the class
- Are declared like regular functions

- Except for two special ones, with no return type
- The constructor:
  - is named like the class
  - is automatically invoked when a variable of the class type is created
- The destructor:
  - is named ***`~classname`***
  - is automatically invoked when a variable of the class type ceases to exist
- Avoid declarations at global scope of objects with non-trivial constructors/destructors
  - There are subtle rules which could reveal deadly

- Methods are commonly defined in a different file

`lfrng.cpp`: Constructor & Destructor

```
#include <cstdlib>
#include <cerrno>
#include "lfrng.h"

using namespace LFRNG;

rng::rng(unsigned n) { // class contructor

    k = l_k;
    l = l_l;
    if (n <= s_k) {
        k = s_k;
        l = s_l;
    } else if (n <= m_k) {
        k = m_k;
        l = m_l;
    } else if (n > l_k)
        errno = EDOM;

    hstr = new unsigned[k];

    init();
}

rng::~rng() {  // class destructor
    delete[] hstr;
}
```

continues on next slide...

`lfrng.cpp`: Initialization & Draw

... follows from previous slide

```
void rng::init() {
    int i;

    for(i=0; i<k; ++i)
        hstr[k-i-1] = rand();

    imk = k-1;
    iml = l-1;
}

unsigned rng::draw() {
    unsigned r;

    r = hstr[imk] + hstr[iml];
    hstr[imk] = r;
    if (imk-- == 0) imk = k-1;
    if (iml-- == 0) iml = k-1;

    return r;
}
```

Methods Definition

- Method definition must be qualified with the class it belongs to
- Being in the class scope, it can access all members without qualification
- The constructor:
  - initializes lags and indexes
  - then allocates the history array
  - Note: allocation failure management is deferred to the user through exception catching
- The destructor:
  - deallocates the history array
  - leaves the rest of the deallocation to default rules
- The remaining methods are pretty similar

User Guide

- To control the seed for initialization

```
srand(my_seed);
```

- To instantiate generators[1]:

```
LFRNG::rng myrgen(68);

using namespace LFRNG;
rng lrgen(98);
rng srgen(55);

rng *rgp;
rgp = new rng(55);
```

- To generate random numbers:

```
unsigned u1, u2;

u1 = myrgen.draw();
u2 = rgp->draw();
```

[1] Did you notice that, unlike in C, `typedef`s are not needed?

Hands-on Session #1

- Write a simple test program that verifies some properties of the generator (e.g. the average)

- Then try a few variations not covered by the User Guide
  - Instantiate a generator like this:

    ```
    LFRNG::rng whatrgen;
    ```

  - Instantiate two generators and assign one to the other
  - Pass a generator by value to a function
  - Try something like this:

    ```
    LFRNG::rng gen;
    gen = 7;
    ```

    or like:

    ```
    LFRNG::rng g9 = 9;
    ```

  - Use a generator for a while and then call its `init()` method

- Carefully recording what happens and your feelings

A Few Thoughts About Initialization

- Is **`init()`** necessary?
  - Yes, it's needed by the constructor
  - No, initialization is already performed by the constructor
  - No, accidental reinitialization of a generator in use could be dangerous
- As a matter of fact, **`init()`** is a C remnant
  - In good C++, initialization is usually completely delegated to constructors
  - Re-initialization can still be performed by destroying and constructing again
- It would however be nice to initialize from an array of seeds, insted of using **`rand()`** to generate them
- Time for refactoring

`lfrng.cpp` Refactored

```
#include <cstdlib>
#include <cerrno>
#include <cstring>
#include "lfrng.h"

using namespace LFRNG;

void rng::build(unsigned n) {  // initializes lags and indexes, allocates history arra

    k = l_k;  l = l_l;
    if (n <= s_k) {
        k = s_k;  l = s_l;
    } else if (n <= m_k) {
        k = m_k;  l = m_l;
    } else if (n > l_k)
        errno = EDOM;

    hstr = new unsigned[k];
}

void rng::random_init() {  // initializes history using rand()
    for(int i=0; i<k; ++i)
        hstr[k-i-1] = rand();

    imk = k-1;  iml = l-1;
}

void rng::array_init(const unsigned *a) {  // initializes history from another array
    memcpy(hstr, a, k*sizeof(unsigned));
    imk = k-1;  iml = l-1;
}
```

`lfrng.h` Refactored

```
#ifndef LFRNG_H
#define LFRNG_H
#include <stdexcept>

namespace LFRNG {

class rng {
    unsigned k, l;
    unsigned imk, iml;
    unsigned *hstr;

    const static unsigned l_k = 98;
    const static unsigned l_l = 27;
    const static unsigned m_k = 73;
    const static unsigned m_l = 31;
    const static unsigned s_k = 55;
    const static unsigned s_l = 24;

    void build(unsigned n);
    void random_init();
    void array_init(const unsigned *a);
public:
    rng(unsigned n) { build(n); random_init(); }
    rng(unsigned n, const unsigned *a) {
        build(n);
        if (n==k) array_init(a);
        else throw std::invalid_argument("unsupported length");
    }
    ~rng();
    unsigned draw();
};
} //namespace LFRNG
#endif
```

`rng` Class Refactored

- The new methods are made private
  - So they are only accessible to other class methods
- Yes, methods can be defined inside the class definition
  - Usually done for short ones (and are `inline`)
- `~rng()` definition is better kept with `build()` definition
  - The `new` in the latter matches `delete` in the former
- Yes, constructors can be overloaded
- When initializing from an array, we'd better be careful
  - A size mismatch is dangerous
  - In a constructor, throwing an exception is much better than anything else
- `throw` throws a value of class type
  - In real life, we'd define exception classes specific to `LFRNG::rng`
  - Let's use a standard one here for simplicity

Default Constructor

- A constructor taking no arguments is termed a *default constructor*
- If you define a class with no constructors, you get a bonus, implicitly defined default constructor
  - It's free, and does next to nothing: call the default constructor of each data member
  - In this case, it wouldn't initialize lags nor allocate the history array
  - Thus, we could accidentally use an uninitialized generator
  - And when the object is destroyed **`delete`** would cause an error
- But a default constructor is good for quick, casual use
  - Let's err on the safe side: let it build the longest supported generator
- Do we have to write yet another constructor?
  - Not really, in this case

lfrng.h No Default Constructor

```
#ifndef LFRNG_H
#define LFRNG_H
#include <stdexcept>

namespace LFRNG {

class rng {
    unsigned k, l;
    unsigned imk, iml;
    unsigned *hstr;

    const static unsigned l_k = 98;
    const static unsigned l_l = 27;
    const static unsigned m_k = 73;
    const static unsigned m_l = 31;
    const static unsigned s_k = 55;
    const static unsigned s_l = 24;

    void build(unsigned n);
    void random_init();
    void array_init(const unsigned *a);
public:
    rng(unsigned n = 98) { build(n); random_init(); }
    rng(unsigned n, const unsigned *a) {
        build(n);
        if (n==k) array_init(a);
        else throw std::invalid_argument("unsupported length");
    }
    ~rng();
    unsigned draw();
};
} //namespace LFRNG
#endif
```

Let’s Use Default Arguments

- We simply provide a default value for the argument in the declaration
- Remember the obvious limitation:
  - If one argument has a default value, all arguments possibly following it must have one too

- We could similarly ‘merge’ the two constructors:
  - giving **`a`** a NULL pointer as default value
  - and initializing with **`random_init()`** if **`a`** is NULL
- But this would be a confusing merge of two different functions, and could slow down construction
- Use default arguments only where they make sense

Beware of Implicit Conversions

- What happens in the following code excerpt?

```
LFRNG::rng gen(98);
gen = 16;
```

  - Objects can be used in expressions, like any other type
  - Implicit type conversions can take place in expressions
  - Constructors with a single argument can also be used for implicit conversions

- Thus the compiler converts the above code into:

```
LFRNG::rng gen(98);
{ LFRNG::rng tmp(16);
   gen = tmp;}
```

- We certainly don't want this absurdity!
- Let's forbid implicit calls to the constructor by making it **`explicit`**

lfrng.h No Implicit Conversions

```
class rng {
    unsigned k, l;
    unsigned imk, iml;
    unsigned *hstr;

    const static unsigned l_k = 98;
    const static unsigned l_l = 27;
    const static unsigned m_k = 73;
    const static unsigned m_l = 31;
    const static unsigned s_k = 55;
    const static unsigned s_l = 24;

    void build(unsigned n);
    void random_init();
    void array_init(const unsigned *a);

  public:
    explicit rng(unsigned n = 98) { build(n); random_init(); }
    rng(unsigned n, const unsigned *a) {
        build(n);
        if (n==k) array_init(a);
        else throw std::invalid_argument("unsupported length");
    }
    ~rng();
    unsigned draw();

};
```

Default Copy and Assignment

- By defining a class you get two more ‘gifts’
- A default *copy constructor*:
  - builds an instance from another object of the class
  - by memberwise copy
  - it’s a necessity to pass objects by value in function calls
- A default = *assignment operator*:
  - performs a memberwise copy
  - it’s a necessity to support objects assignments

Default Copy and Assignment

- When a data member is a pointer, memberwise copy is said to be *shallow copy*

```
rng r1;
rng r2 = r1;  // call copy constructor: trouble here
rng r3;

r3 = r2;      // call copy assignment: trouble here
```

  - May cause memory leaks overwriting the previous pointer content
  - May cause double deletion of the same memory area in destructors (a fatal error)

- We need to explicitly define *deep* copy constructor and assignment

`lfrng.h` Deep Copies

```
class rng {
    unsigned k, l;
    unsigned imk, iml;
    unsigned *hstr;

    const static unsigned l_k = 98;
    const static unsigned l_l = 27;
    const static unsigned m_k = 73;
    const static unsigned m_l = 31;
    const static unsigned s_k = 55;
    const static unsigned s_l = 24;

    void build(unsigned n);
    void random_init();
    void array_init(const unsigned *a);
    void copy_in(const rng& g);

  public:
    explicit rng(unsigned n = 98) { build(n); random_init(); }
    rng(unsigned n, const unsigned *a) {
        build(n);
        if (n==k) array_init(a);
        else throw std::invalid_argument("unsupported length");
    }
    ~rng();
    unsigned draw();

    rng(const rng& g) { copy_in(g); }   // copy constructor
    rng& operator= (const rng& g);       // copy assignment
};
```

Implementing Deep Copies

- The combination of reference and **`const`** arguments in copy constructor and assignment operator is mandatory

- Copy construction and assignment have much in common
- But one big difference:
  - the left operand of the assignment operator must already exist
  - thus it contains an already allocated history array, which should be deleted first
- But what about **`g = g`**?
  - It's perfectly legal!
  - And we'd better not delete the history array in that case!

- **`this`** it's a reserved keyword, the address of the object the method was invoked on
  - For the assignment operator, its left operand

Adding Deep Copy to `lfrng.cpp`

(Includes and previously defined methods unchanged)

```
void rng::copy_in(const rng& g) {
    k = g.k;
    l = g.l;
    hstr = new unsigned[k];
    memcpy(hstr, g.hstr, k*sizeof(unsigned));
    imk = g.imk;
    iml = g.iml;
}

rng& rng::operator= (const rng& g) {
    if (this != &g) {
        delete[] hstr;
        copy_in(g);
    }
    return *this;
}
```

Few Thoughts on RNG Copy & Assignment

- They could be unsafe if used without care
  - The same term of the sequence could be used more than once in a simulation
  - We'd better to get rid of them
  - We could make them `private`
- They could be useful if used with care
  - E.g. to compare algorithms
  - Or for very specific algorithms that need the same sequence more than once
  - best reasons are debugging and class specialization
- Let's make them `protected`
  - I.e. only selected classes and functions will be able to access them
  - More on this later

`lfrng.h` Private Deep Copies

```
class rng {
    unsigned k, l;
    unsigned imk, iml;
    unsigned *hstr;

    const static unsigned l_k = 98;
    const static unsigned l_l = 27;
    const static unsigned m_k = 73;
    const static unsigned m_l = 31;
    const static unsigned s_k = 55;
    const static unsigned s_l = 24;

    void build(unsigned n);
    void random_init();
    void array_init(const unsigned *a);
    void copy_in(const rng& g);

    rng(const rng& g) { copy_in(g); }   // copy constructor
    rng& operator= (const rng& g);      // copy assignment
  public:
    explicit rng(unsigned n = 98) { build(n); random_init(); }
    rng(unsigned n, const unsigned *a) {
        build(n);
        if (n==k) array_init(a);
        else throw std::invalid_argument("unsupported length");
    }
    ~rng();
    unsigned draw();

};
```

lfrng.h Protected Deep Copies

```
class rng {
    unsigned k, l;
    unsigned imk, iml;
    unsigned *hstr;

    const static unsigned l_k = 98;
    const static unsigned l_l = 27;
    const static unsigned m_k = 73;
    const static unsigned m_l = 31;
    const static unsigned s_k = 55;
    const static unsigned s_l = 24;

    void build(unsigned n);
    void random_init();
    void array_init(const unsigned *a);
    void copy_in(const rng& g);
  protected:
    rng(const rng& g) { copy_in(g); }   // copy constructor
    rng& operator= (const rng& g);      // copy assignment
  public:
    explicit rng(unsigned n = 98) { build(n); random_init(); }
    rng(unsigned n, const unsigned *a) {
        build(n);
        if (n==k) array_init(a);
        else throw std::invalid_argument("unsupported length");
    }
    ~rng();
    unsigned draw();

};
```

Better Lag Management

- Up to now, we only support three good pairs of lags, which is easy
- But there is a numerable infinity available
- So we could add more in future releases
- Managing them with names is tough and requires code changes

- A sensible plan:
  - Add a static table of lags pairs to the class
  - Parameterize the logic to choose the right one
- We need a base type for this table, but don't want to pollute or cause name clashes

lfrng.h: Table of Lags

```
class rng {
    unsigned k, l;
    unsigned imk, iml;
    unsigned *hstr;

    struct pair {
        unsigned k, l;
        pair(unsigned i, unsigned j) : k(i), l(j) {}
    };
    const static unsigned n_lags = 3;
    const static pair lags[n_lags];

    void build(unsigned n);
    void random_init();
    void array_init(const unsigned *a);
    void copy_in(const rng& g);
protected:
    rng(const rng& g) { copy_in(g); }
    rng& operator= (const rng& g);
public:
    explicit rng(unsigned n = 98) { build(n); random_init(); }
    rng(unsigned n, const unsigned *a) {
        build(n);
        if (n==k) array_init(a);
        else throw std::invalid_argument("unsupported length");
    }
    ~rng();
    unsigned draw();
};
```

Nested Classes & More

- Nested classes are classes defined inside another class
  - Only visible in the enclosing class scope
  - Good for local utilities
- Initialization of data members:
  - is better performed by invoking their constructor directly
  - unless preliminary calculations are needed
- Unfortunately, static array members cannot be initialized inside the class
- We’ll put initialization in `lfrng.cpp`, where we have to change build as well

lfrng.cpp: Table of Lags

```
#include <exception>
#include "lfrng.h"

using namespace LFRNG;

const rng::pair rng::lags[rng::n_lags] = {rng::pair(55,24),
                                                rng::pair(73,31),
                                                rng::pair(98,27)};


void rng::build(unsigned n) {
    int i;

    for(i = 0; i < n_lags; ++i) {
        l = lags[i].l;
        k = lags[i].k;
        if (n <= k) break;
    }
    if (n > k) throw std::invalid_argument("unsupported length");

    hstr = new unsigned[k];
}
```

Other methods follow unchanged

Static & Const Methods

- It would be nice for users to know:
  - maximum length supported by `rng`
  - actual length of a `rng` object
- Let's add two query methods
- Wait! To call `max_len()` we need an instance of the class
  - This is nonsensical
  - Let's make it callable independently
- `static` methods can be called without instantiating the class, like this:

```
unsigned ml = rng::max_len();
```

- `const` methods cannot modify the object

lfrng.h: Table of Lags

```
class rng {
    unsigned k, l;
    unsigned imk, iml;
    unsigned *hstr;

    struct pair {
        unsigned k, l;
        pair(unsigned i, unsigned j) : k(i), l(j) {}
    };
    const static unsigned n_lags = 3;
    const static pair lags[n_lags];

    void build(unsigned n);
    void random_init();
    void array_init(const unsigned *a);
    void copy_in(const rng& g);
protected:
    rng(const rng& g) { copy_in(g); }
    rng& operator= (const rng& g);
public:
    explicit rng(unsigned n = 98) { build(n); random_init(); }
    rng(unsigned n, const unsigned *a) {
        build(n);
        if (n==k) array_init(a);
        else throw std::invalid_argument("unsupported length");
    }
    ~rng();
    static unsigned max_len() { return lags[n_lags-1].k; }
    unsigned len() const { return k; }
    unsigned draw();
};
```

A Final Touch

- Let’s make **`draw()`** method protected
- And use the function call operator **`()`** to draw terms of the sequence

- Thus, if **`g`** is an instance of **`LFRNG::rng`** class, we can draw random numbers like this:

  ```
  i = g();
  ```
- An object like this is termed a *functor*

- We are doing this for two reasons
  - It’s *unbelievably* cool! Isn’t it?
  - Will come useful later on

`lfrng.h`: Table of Lags

```
class rng {
    unsigned k, l;
    unsigned imk, iml;
    unsigned *hstr;

    struct pair {
        unsigned k, l;
        pair(unsigned i, unsigned j) : k(i), l(j) {}
    };
    const static unsigned n_lags = 3;
    const static pair lags[n_lags];

    void build(unsigned n);
    void random_init();
    void array_init(const unsigned *a);
    void copy_in(const rng& g);
protected:
    unsigned draw();
    rng(const rng& g) { copy_in(g); }
    rng& operator= (const rng& g);
public:
    explicit rng(unsigned n = 98) { build(n); random_init(); }
    rng(unsigned n, const unsigned *a) {
        build(n);
        if (n==k) array_init(a);
        else throw std::invalid_argument("unsupported length");
    }
    ~rng();
    static unsigned max_len() { return lags[n_lags-1].k; }
    unsigned len() const { return k; }
    unsigned operator() () { return draw(); }
};
```

Hands-on Session #2

- Time to try the latest and greatest version

- Check all misuses are not allowed anymore

What Objects are Good For?

- Tie together data structures and their manipulating functions
- Protect innards of a data type from inappropriate access
- Hide implementation details
- Automate elaborate initialization and disposal of data structures
- Control in detail what operations can be performed on a data type

- And more...

Outline

1 Do you Need an Object?

2 Inheritance and Polymorphism
- Heads and Tails
- Floating Point RNGs
- Summing it Up

3 Class I/O

A Coin Class

```
#include <limits>

// rng class definition omitted

class coin : public rng {
public:
    explicit coin(unsigned n=98) : rng(n) {}
    coin(unsigned n, const unsigned *a) : rng(n,a) {}
    bool operator() () {
        unsigned h = std::numeric_limits<unsigned>::max()/2;
        return rng::draw() > h;
    }
};
```

Derived and Base Classes

- `LFRNG::coin` is a derived class of `LFRNG::rng`, i.e.:
  - inherits all `rng` members
  - may ovverride them or add new ones
  - has access to public and protected `rng` members

- `rng` is a `public` base class of `coin`:
  - all `rng` public members (like `max_len()` or `len()`) are accessible through `coin`
  - classes derived from `coin` have access to `rng` protected members
- Were `rng` a `protected` base class of `coin`:
  - only `coin` methods and classes derived from `coin` would have access to `rng` public and protected members
- Were `rng` a `private` base class of `coin`:
  - only `coin` has access to `rng` public and protected members

Constructors & Destructors in Derivation

- Base class constructor must be invoked:
  - *before* constructing data members possibly added in the derived class
  - between a `:` and the derived class constructor body
- Common mistake: should you write

  `coin(unsigned n) {};`

  the base class constructor would still be implicitly invoked first, not the one you want however!
- Destructors:
  - take no parameters, so implicit invocation is ok
  - are invoked in the opposite order
- As we added no data members in `coin`, the bonus default destructor is all we need

Methods Override

- The `coin` class has its own constructors and destructors
- `max_len()` and `len()` are the base class ones
- `()` operator is overridden to do the right thing
  - draw a random unsigned integer using its base class protected method *draw()*
  - converting it to a `bool` according to which half of its range it falls into

- By the way:
  - `limits` is the C++ header providing info on integer and floating point types
  - in form of static methods of special purpose classes
  - `std::numeric_limits<`*`type`*`>` is a template class (guess what, we'll learn more later)
- Good ol' C defines are provided in the `climits` header to ease conversion, but avoid them in new codes

Hands-on Session #3

- Toss the coin
- Derive from `LFRNG::rng` two classes to generate *odd* and *even* random numbers
- Derive from `LFRNG::rng` a bingo class:
  - returning integers from 1 to 90
  - each of them once
  - providing useful utility functions
  - with reasonable behavior when extractions are over
- Hint:
  1. set m to 90
  2. initialize an array with integers from 1 to 90
  3. generate a random index $i : 0 \leq i < m$
  4. swap i-th and m-th elements of the array
  5. return the m-th element of the array
  6. set m to $m - 1$
  7. if $m > 0$ goto 3

Floating Point RNGs

- We need a floating point RNG and want to reuse **`LFRNG::rng`**, which is tested and tried
- Coins, odd and even RNGs, bingos, are special cases of an integer RNG (*isA* relationship)
- A floating point RNG is not, for a number of reasons
  - FP numbers mimic real numbers, which are a superset of integers, not a subset
  - Lagged Fibonacci is not the best RNG in the world, we may possibly have to change in the future
  - Other fast and very good floating point generators like AWC or SWB are available
- We'll not derive from **`LFRNG::rng`**, will use the latter as a member of the new class (*hasA* relationship)

frng.h

```
#ifndef FRNG_H
#define FRNG_H

#include <limits>
#include "lfrng.h"

namespace FPRNG {

class frng {
    LFRNG::rng intgen;
public:
    explicit frng(unsigned n = 98) : intgen(n) {}
    frng(unsigned n, const unsigned *a): intgen(n, a) {}
    unsigned len() { return intgen.len(); }
    static unsigned max_len() { return LFRNG::rng::max_len(); }
    double operator() () {
         double m = std::numeric_limits<unsigned>::max();
         return intgen()/m;
    }
};

} // namespace FPRNG

#endif
```

Member Classes Construction

- Data members are constructed like base classes
- Except that member name is used instead of class name

- As with base classes, members constructors can be implicitly called
- Common mistake: writing

  ```
  class foo {bar b; public: foo(bar inb) {b = inb; }};
  ```

  which is equivalent to:

  ```
  class foo {bar b; public: foo(bar inb) : b() {b = inb; }};
  ```

- For native types, this is irrelevant, for classes this could double the cost of costruction of each member

Looking for More Flexibility

- This solution is rigid
- **`frng`** generates according to a uniform distribution
- Many distributions are available and useful
- Moreover, we want to write some algorithms (like Montecarlo integrators) independently from the actual distribution of the RNG
- Again, class derivation comes to the rescue

Enter Polymorphism

- In C++, pointers and references to a base class can point/refer to a derived class
- Of course, if a method is invoked on the pointer/reference, it will be the one of the base class
- Unless the method was made **`virtual`**, in which case the one of the actual object class will be called
- More flexibility at a cost: consulting tables of addresses in memory

- Access to polymorphism can be controlled:
  - for **`public`** base classes, polymorphism is available to any function
  - for **`protected`** base classes, polymorphism is available only to the derived classes and its descendants
  - for **`private`** base classes, polymorphism is available only to the derived class

Implementing Polymorphism

- Let’s add to `frng` a protected `draw()` method
  - It bridges the gap with the underlying, private generator
- Let’s make the `()` method a virtual function
- Let’s make it a *pure* virtual function by ’assigning’ `0` to it
- This makes `frng` an abstract class, i.e. no object can be instantiated
  - We only need it for pointers and references
- Now let’s add the `furng` class
  - Which has nothing special, except the virtual method is not pure
- But to realize the power of polymorphism, we need more RNGs

frng.h: Polymorphism

```
#ifndef FRNG_H
#define FRNG_H

#include <limits>
#include "lfrng.h"

namespace FPRNG {

class frng {                 // generic FP RNG
    LFRNG::rng intgen;
protected:
    double draw() {
         double m = std::numeric_limits<unsigned>::max()
         return intgen()/m;
         }
public:
    explicit frng(unsigned n = 98) : intgen(n) {}
    frng(unsigned n, const unsigned *a): intgen(n, a) {}
    unsigned len() { return intgen.len(); }
    static unsigned max_len() { return LFRNG::rng::max_len(); }
    virtual double operator() () = 0;
};

class furng : public frng {  // uniform FP RNG in [0,1)
public:
    explicit furng(unsigned n = 98) : frng(n) {}
    furng(unsigned n, const unsigned *a): frng(n, a) {}
    virtual double operator() () { return frng::draw(); }
};
} // namespace FPRNG

#endif
```

`frng.h`: More RNGs

```
class fsurng : public frng {                       // scaled uniform FP RNG
    double offset, scale;
public:
    fsurng(double o, double s, unsigned n = 98) : offset(o), scale(s), frng(n) {}
    fsurng(unsigned n, const unsigned *a): frng(n, a) {}
    virtual double operator() () { return frng::draw()*scale + offset; }
};

class ferng : public frng {                        // exponential FP RNG
public:
    explicit ferng(unsigned n = 98) : frng(n) {}
    ferng(unsigned n, const unsigned *a): frng(n, a) {}
    virtual double operator() ();
};

class fnrng : public frng {                        // normal FP RNG
    const static double pi2 = 2.0*3.1415926535897932384626433832795;
    double ndr;
    bool cached;
public:
    explicit fnrng(unsigned n = 98) : cached(false), frng(n) {}
    fnrng(unsigned n, const unsigned *a): cached(false), frng(n, a) {}
    virtual double operator() ();
};
```

`frng.cpp`: More RNGs

```
#include <cmath>
#include "frng.h"

using namespace FPRNG;

double ferng::operator() () {             // exponentially distributed
    double r;
    while(0.0 == (r = frng::draw()));
    return -log(r);
}

double fnrng::operator() () {             // normally distributed
    double x1, x2, r2, f;

    if (cached) {
        cached = false;
        return ndr;
    }

    do {
        x1 = frng::draw()*2.0 - 1.0;
        x2 = frng::draw()*2.0 - 1.0;
        r2 = x1*x1 + x2*x2;
    } while(r2 > 1.0 || 0.0 == r2);
    f = sqrt(-2.0*log(r2)/r2);
    ndr = x2*f;
    cached = true;
    return x1*f;
};
```

Hands-on Session #4

- Let's experiment how it works
- Try to instantiate and use all FP generator classes (**`frng`** too!)
- Write a function:
  - accepting an **`frng`** pointer or reference as argument
  - exercising it to compute average, variance or some other moment
- Test with all the generators we defined

More Polymorphism

- A derived class can be abstract too
- And a protected method can be virtual too

- Let's write a generic rejecton RNG class
- Basic idea of rejection generation
    - you have a PDF $f(x)$ mapping $[a, b)$ to $[0, P)$
    - randomly generate x_i uniformly distributed in $[a, b)$
    - randomly generate x_{i+1} uniformly distributed in $[0, P)$
    - if $x_{i+1} < f(x_i)$ then return x_i and throw x_{i+1} away
    - otherwise throw away both and retry

- Then let's derive from it a generator with a triangle distribution in $[-1, 1)$

frng.h: Adding Rejection RNGs

```
class frejrng : public frng {                 // rejection method RNGs abstract base
protected:
    virtual bool accept(double u1, double u2, double& r) = 0;
public:
    explicit frejrng(unsigned n = 98) : frng(n) {}
    frejrng(unsigned n, const unsigned *a): frng(n, a) {}
    double operator() ();
};

class ftrianglerng : public frejrng {
protected:
    virtual bool accept(double u1, double u2, double& r);
public:
    explicit ftrianglerng(unsigned n = 98) : frejrng(n) {}
    ftrianglerng(unsigned n, const unsigned *a): frejrng(n, a) {}
};
```

`frng.cpp`: Adding Rejection RNGs

```
double frejrng::operator() () {
    double r;
    while(!accept(frng::draw(), frng::draw(), r));
    return r;
}

bool ftrianglerng::accept(double u1, double u2, double& r) {
    r = u1*2.0 - 1.0;
    if ( u2 > (1.0 - fabs(r)) )
        return false;
    return true;
};
```

Hands-on Session #5

- Test it
- Then derive another for the distribution:

$$p(x) = \begin{cases} \frac{3}{2}x^2 & x \in [-1, 1) \\ 0 & \text{otherwise} \end{cases}$$

- Or for a different distribution of your choice

What Inheritance is Good For?

- To reuse code without rewriting it
- To properly differentiate behavior of similar classes in a robust way
- To define methods that derived classes must implement
- To write functions that can operate on objects of different classes in the same hierarchy
- To control in detail where polymorphism is allowed

- And more...

- A *caveat*: if you are concerned with performances, polymorphism could impact them

Outline

1. Do you Need an Object?
2. Inheritance and Polymorphism
3. Class I/O
   - Basics
   - Inheritance and I/O

User Defined I/O

- Actually quite simple
  - Just write overloaded versions of `<<` and `>>`
  - And make them `rng` friends
- A member function declaration specifies three logically distinct things:
  - the function can access the private part of class declaration
  - the function is in the scope of the class
  - the function must be invoked on an object (has a *this* pointer)
- By declaring a member function *static*, we get the first twos
- By declaring a function as a *friend*, we get only the first

User Defined I/O for **rng**

- So, let's add to `rng` class the declarations:

```
friend ostream& operator<< (ostream& s, const rng& g);
friend istream& operator>> (istream& s, rng& g);
```

- Write them for `ostream` and `istream` respectively
  - All others streams of interest inherit from them
- Beware: `rng` class definition is in `LFRNG` namespace
  - All member declarations are in the same namespace
  - You don't need to explicitly put their definitions in it
  - The `rng::` scope resolution in their definitions is enough
  - Friends are not members!
  - Their definitions must be explicitly put in the the namespace

Managing Failures

- The really important thing is to correctly address failures
- Easy for output
  - The object state doesn’t change
  - Failure and bad state are preserved by next operations
- Crucial for input
  - The object state will change
  - And we want the new one to be consistent
- Possible source of input errors:
  1. read of an `rng` member fails
  2. lags read from the stream differ from the ones already stored in the object
- For ease of use, it is of paramount importance that the specialized `>>` version behaves consistently with Standard Library versions

rng::operator<<

```
std::ostream& operator<< (std::ostream& s, const rng& g) {
    int i;

    s << g.k << ' ' << g.l << std::endl;
    s << g.imk << ' ' << g.iml << std::endl;
    for(i = 0; i<g.k; ++i)
        s << g.hstr[i] << ' ';
    s << std::endl;

    return s;
}
```

rng::operator>>

```
std::istream& operator>> (std::istream& s, rng& g) {
    unsigned k, l, imk, iml;
    unsigned *hstr;
    k = l = 0;

    s >> k >> l;
    if (k != g.k || l != g.l) {
        s.clear(std::ios_base::failbit);
        return s;
    } else {
        hstr = new unsigned[k];
        s >> imk >> iml;
        for(int i = 0; i<k; ++i)
            s >> hstr[i];
    }
    if (s) {
        g.k = k;
        g.l = l;
        g.imk = imk;
        g.iml = iml;
        memcpy(g.hstr, hstr, k*sizeof(unsigned));
    }
    delete[] hstr;
    return s;
}
```

Managing Input Failures

- We first read in the lags
- By design, the object is alredy initialized so the lags must match
- If they don't, we fail
    - By setting the stream fail state bit and returning
    - **`s.clear()`** actually sets the state, very intuitive name!
- Otherwise, we read in the generator recent history in temporary areas
- Eventually, we get rid of temporary storage

In Real Life

- We are not managing **`new`** exceptions, we'd better:

```
try {
    hstr = new unsigned[k];
} catch (...) {      // catch any exception
    s.clear(std::ios_base::failbit);
    throw;           // re-throw the catched exception
}
```

- It is improbable for a **`rng`** to be input by keyboard
- But a file could be changed by mistake
- We'd better:
  - add a prolog and epilog string like **`"LFRNG::rng"`** in output
  - and check for both on input
  - and output a good checksum too
  - to be verified on input

Hands-on Session #6

- Get back at the xyz-format exercise
- Define a `class` for data of a single atom
- And overload I/O operators for it
- Once again, check you correctly managed exceptions using:
  - file names that do not exist
  - files in the wrong format
  - files with missing data
- Homework assignment: building on the above `class`,
  - define a `class` to hold all data from an xyz-format file
  - independently of the number of atoms
  - and write consistent I/O operators for them

I/O for **`rng`** Derived Classes

- For **`coin`**, nothing to do
  - A derived class can be implicitly converted to its base class
  - **`rng`** overloaded I/O operators will match it
  - They are ok, as **`coin`** doesn't define new data members
- Things are different if we add or redefine data members
- Let's imagine that for a really insane reason, we don't want to get the first random number again
  - Let's derive a **`nofirst`** class from **`rng`**
  - throwing an exception if the first one is drawn again

nofirst Class

```
class nofirst : public rng {
    unsigned first;
    bool takeit;
public:
    struct first_twice : public std::runtime_error {
        first_twice(const first_twice& e) : std::runtime_error(e) {}
        first_twice(const char *s) : std::runtime_error(s) {}
    };

    explicit nofirst(unsigned n=98) : rng(n), takeit(true) {}
    nofirst(unsigned n, const unsigned *a) : rng(n,a), takeit(true) {}

    unsigned operator() () {
        unsigned next = rng::draw();
        if (takeit) {
            first = next;
            takeit = false;
        } else if (next == first)
            throw first_twice("first one occurred again");
        return next;
    }

    friend std::ostream& operator<< (std::ostream& s, const nofirst& g);
    friend std::istream& operator>> (std::istream& s, nofirst& g);
};
```

`nofirst` Remarks

- Exceptions are classes
- If an exception is very specific, it's better to define a specific class
- Inheriting from standard ones makes it easy, but not mandatory
- We can now **`catch LFRNG::nofirst::wrap`**

- We added data members
- Thus we have to specialize I/O operators
  - They'll invoke the base class one
  - Then care of **`nofirst`** specific stuff

nofirst I/O Operators

```
std::ostream& operator<< (std::ostream& s, const nofirst& g) {

    return s << static_cast<const rng&>(g)
             << g.takeit << ' ' << g.first << std::endl;
}



std::istream& operator>> (std::istream& s, nofirst& g) {
    nofirst temp(g);

    s >> static_cast<rng&>(temp);
    if (s)
        s >> temp.takeit >> temp.first;
    if (s)
        g = temp;
    return s;
}
```

Safety First

- To invoke base class operators, we must cast to base class references
  - Otherwise, the operator would recursively call itself
- Cast of pointers and references is dangerous
- And should be limited to controlled places
  - Like member and friend functions
- C casts do not allow safety checks: strongly discouraged!
- C++ **`static_cast<>`** allows for some compiler checks
  - Like forbid casting **`const`** references to non-**`const`** ones

- We have to use a temporary to change the object only when all I/O succeded
  - Our protected copy constructor and assignment found a proper use

Hands-on Session #7

- Easy: test that I/O operators work on **`rng`** and its descendants

I/O for `frng` Inheritance Tree

- Easy, if you don't support runtime polymorphism in I/O
  - Add to `frng` and descendants the protected copy constructors and assignments we dispensed with for simplicity
  - Write friend overloaded I/O operators for `frng`
  - They simply read/write its `rng` member, `intgen`
  - And will also work for `furng`, `ferng`, `frejrng`, and `ftrianglerng`
  - Then overload them for descendants adding data members

- If you need polymorphic I/O in a function accepting any `frng` descendant, it's a different story
  - Make `frng` class a friend of `rng` class
  - Add to `frng` two `virtual` methods: `read()` and `write()`
  - Make `frng` I/O operators defer all actual I/O to them
  - Then simply override `read()` and `write()` for descendants adding data members

Polymorphic I/O

```
void frng::write(std::ostream& s) const {

    s << intgen;
}

void frng::read(std::istream& s) {
    LFRNG::rng temp(this->intgen);

    s >> temp;
    if (s)
        this->intgen = temp;
}

std::ostream& operator<< (std::ostream& s, const frng& g) {

    g.write(s);
    return s;
}

std::istream& operator>> (std::istream& s, frng& g) {

    g.read(s);
    return s;
}
```

Homework Assignment

- Override **`read()`** and **`write()`** virtual methods in
  - **`fsurng`** class
  - **`fnrng`** class

- Their overridden versions must be modeled on **`nofirst`** I/O operators
- But you have to use **`dynamic_cast<>`** for casting
  - Much like *static_cast<>*
  - But adds runtime safety checks

- No need to overload **`frng`** I/O operators
- That’s the beauty of runtime polymorphism!

Strict Formatting Requirements

- `frng` descendants add floating point data members
- Exact translation requires a minimum precision
  - Like 9 digits for `floats`
  - And 19 digits for `doubles`
  - Default precision (6 digits) is a bad mistake
- You must enforce it inside overridden I/O functions
  - surrounding I/O operations might need a different one
  - deferring issue to users is error prone and annoying

- Beware! formatting state is stateful on streams
- You'd better save it beforehand:

```
ios_base::fmtflags savefmt = s.flags();
```

to restore it when you are done:

```
s.flags(savefmt);
```

Rights & Credits

Slides and examples were authored by:

- Michela Botti
- Federico Massaioli
- Luca Ferraro
- Stefano Tagliaventi