Standard Template Library

Victor Eijkhout, Susan Lindsey

Fall 2022

`last formatted: November 29, 2022`

1. Standard Template Library

- C++ is language syntax plus STL:
  headers such as `vector`
- Some people (read: large companies) write their own STL.
- Here are some useful bits from the STL;
  there are many more.

Random number generation

2. What are random numbers?

- Not really random, just very unpredictable.
- Often based on integer sequences:

$$r_{n+1} = ar_n + b \mod N$$

- $\Rightarrow$ they repeat, but only with a long period.
- A good generator passes statistical tests.

3. Random generators and distributions

- Random device

```
std::default_random_engine generator;
% random seed:
std::random_device r;
std::default_random_engine generator{ r() };
```

- Distributions:

```
std::uniform_real_distribution<float> distribution(0.,1.);
std::uniform_int_distribution<int> distribution(1,6);
```

- Sample from the distribution:

```
std::default_random_engine generator;
std::uniform_int_distribution<> distribution(0,nbuckets-1);
random_number = distribution(generator);
```

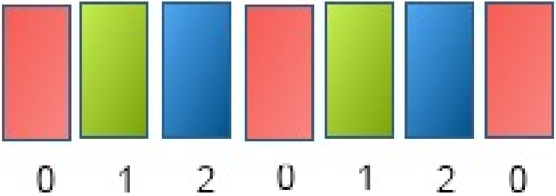
- Do not use the old C-style `random`!

4. Why so complicated?

- Large period wanted; C random has 2^{15}.
- Multiple generators, guarantee on quality.
- Simple transforms have a bias:

```c
int under100 = rand() % 100
```

Simple example: period 7, mod 3



5. Dice throw

```
// set the default generator
std::default_random_engine generator;

// distribution: ints 1..6
std::uniform_int_distribution<int> distribution(1,6);

// apply distribution to generator:
int dice_roll = distribution(generator);
  // generates number in the range 1..6
```

6. Poisson distribution

Another distribution is the Poisson distribution:

```
std::default_random_engine generator;
float mean = 3.5;
std::poisson_distribution<int> distribution(mean);
int number = distribution(generator);
```

7. Global engine

Wrong approach:

Code:

```
int nonrandom_int(int max) {
  std::default_random_engine engine;
  std::uniform_int_distribution<>
     ints(1,max);
  return ints(engine);
};
```

Output:

```
Three ints: 15, 15,
     15.
```

Good approach:

Code:

```
int realrandom_int(int max) {
  static
     std::default_random_engine
     static_engine;
  std::uniform_int_distribution<>
     ints(1,max);
  return ints(static_engine);
```

Output:

```
Three ints: 15, 98,
     70.
```

8. Generator in a class

Note the use of `static`:

```
class generate {
private:
  static inline std::default_random_engine engine;
public:
  static int random_int(int max) {
    std::uniform_int_distribution<> ints(1,max);
    return ints(generate::engine);
  };
};
```

Usage:

```
auto nonzero_pcnt = generate::random_int(100)
```

Time

9. Chrono

```
#include <chrono>

// several clocks
using myclock = std::chrono::high_resolution_clock;

// time and duration
auto start_time = myclock::now();
auto duration = myclock::now()-start_time;
auto microsec_duration =
    std::chrono::duration_cast<std::chrono::microseconds>
                (duration);
cout << "This took "
     << microsec_duration.count() << "usec\n"
```

More

10. Complex numbers

```
#include <complex>

complex<float> f;
f.re = 1.; f.im = 2.;
complex<double> d(1.,3.);

using std::complex_literals::i;
std::complex<double> c = 1.0 + 1i;

conj(c); exp(c);
```

11. Example usage

Code:

```
vector< complex<double> > vec1(N,
     1.+2.5i );
auto vec2( vec1 );
/* ... */
for ( int i=0; i<vec1.size(); i++ )
     {
  vec2[i] = vec1[i] * ( 1.+1.i );
}
/* ... */
auto sum = accumulate
  ( vec2.begin(),vec2.end(),
    complex<double>(0.) );
cout << "result: " << sum << '\n';
```

Output:

```
result:
     (-1.5e+06,3.5e+06)
```

Tuples; Union-like stuff

12. C++11 style tuples

```
#include <tuple>

std::tuple<int,double,char> id = \
    std::make_tuple<int,double,char>( 3, 5.12, 'f' );
    // or:
    std::make_tuple( 3, 5.12, 'f' );
double result = std::get<1>(id);
std::get<0>(id) += 1;

// also:
std::pair<int,char> ic =
  make_pair( 24, 'd' );
```

Annoyance: all that 'get'ting.

13. Function returning tuple

Return type deduction:

```
#include <tuple>
using std::make_tuple,
    std::tuple;
  /* ... */
auto maybe_root1(float x) {
  if (x<0)
    return make_tuple
      <bool,float>(false,-1);
  else
    return make_tuple

     <bool,float>(true,sqrt(x));
};
```

Alternative:

```
tuple<bool,float>
    maybe_root2(float x) {
  if (x<0)
    return {false,-1};
  else
    return {true,sqrt(x)};
};
```

14. Catching a returned tuple

The calling code is particularly elegant:

Code:

```
auto [succeed,y] = maybe_root2(x);
if (succeed)
  cout << "Root of " << x
       << " is " << y << '\n';
else
  cout << "Sorry, " << x
       << " is negative" << '\n';
```

Output:

```
Root of 2 is 1.41421
Sorry, -2 is negative
```

This is known as structured binding.

15. Returning two things

simple solution:

```
bool RootOrError(float &x) {
  if (x<0)
    return false;
  else
    x = sqrt(x);
  return true;
};
  /* ... */
  for ( auto x : {2.f,-2.f} )
    if (RootOrError(x))
      cout << "Root is " << x << '\n';
    else
      cout << "could not take root of " << x << '\n';
```

other solution: tuples

16. Tuple solution

```
#include <tuple>
using std::tuple, std::pair;
  /* ... */
pair<bool,float> RootAndValid(float x) {
  if (x<0)
    return {false,x};
  else
    return {true,sqrt(x)};
};
  /* ... */
  for ( auto x : {2.f,-2.f} )
    if ( auto [ok,root] = RootAndValid(x) ; ok )
      cout << "Root is " << root << '\n';
    else
      cout << "could not take root of " << x << '\n';
```

Variants

17. More variant methods

```
variant<int,double,string> union_ids;

union_ids = 3.5;
switch ( union_ids.index() ) {
case 1 :
  cout << "Double case: " << std::get<double>(union_ids) << '\n';
}

union_ids = "Hello world";
if ( auto union_int = get_if<int>(&union_ids) ; union_int )
  cout << "Int: " << *union_int << '\n';
else if ( auto union_string = get_if<string>(&union_ids) ; union_string
     )
  cout << "String: " << *union_string << '\n';
```

Exercise 1

Write a routine that computes the roots of the quadratic equation

$$ax^2 + bx + c = 0.$$

The routine should return two roots, or one root, or an indication that the equation has no solutions.

Code:

```
for ( auto coefficients :
        { make_tuple(2.0, 1.5,
    2.5),
          make_tuple(1.0, 4.0,
    4.0),
          make_tuple(2.2, 5.1,
    2.5)
        } ) {
  auto result =
    compute_roots(coefficients);
```

Output:

```
With a=2 b=1.5 c=2.5
No root
With a=2.2 b=5.1
    c=2.5
Root1: -0.703978
    root2: -1.6142
With a=1 b=4 c=4
Single root: -2
```

18. Problem setup

Represent the polynomial

$$ax^2 + bx + c$$

as

```
using quadratic = tuple<double,double,double>;
```

Unpack:

```
auto [a,b,c] = coefficients;
```

`assert` something here?

Exercise 2

Write a function

```
double discriminant( quadratic coefficients );
```

that computes $b^2 - 4ac$, and test:

```
TEST_CASE( "discriminant" ) {
  REQUIRE( discriminant( make_tuple(0., 2.5, 0.) )
     ==Catch::Approx(6.25) );
  REQUIRE( discriminant( make_tuple(1., 0., 1.5 ) )
     ==Catch::Approx(-6.) );
  REQUIRE( discriminant( make_tuple(.1, .1, .1*.5 ) )
     ==Catch::Approx(-.01) );
}
```

Exercise 3

Write a function

```
bool discriminant_zero( quadratic coefficients );
```

that passes the test

```
quadratic coefficients = make_tuple(a,b,c);
d = discriminant( coefficients );
z = discriminant_zero( coefficients );
INFO( a << "," << b << "," << c << " d=" << d );
REQUIRE( z );
```

Using for instance the values:

```
a = 2; b = 4; c = 2;
a = 2; b = sqrt(40); c = 5; // !!!
a = 3; b = 0; c = 0.;
```

Exercise 4

Write the function *simple_root* that returns the single root. For confirmation, test

```
auto r = simple_root(coefficients);
REQUIRE( evaluate(coefficients,r)==Catch::Approx(0.).margin(1.e-14) );
```

Exercise 5

Write a function that returns the two roots as a indexcstdpair:

```
pair<double,double> double_root( quadratic coefficients );
```

Test:

```
quadratic coefficients = make_tuple(a,b,c);
auto [r1,r2] = double_root(coefficients);
auto
  e1 = evaluate(coefficients,r1),
  e2 = evaluate(coefficients,r2);
REQUIRE( evaluate(coefficients,r1)==Catch::Approx(0.).margin(1.e-14) );
REQUIRE( evaluate(coefficients,r2)==Catch::Approx(0.).margin(1.e-14) );
```

Exercise 6

Write a function

```
variant< bool,double, pair<double,double> >
    compute_roots( quadratic coefficients);
```

Test:

```
TEST_CASE( "full test" ) {
  double a,b,c; int index;
  SECTION( "no root" ) {
    a=2.0; b=1.5; c=2.5;
    index = 0;
  }
  SECTION( "single root" ) {
    a=1.0; b=4.0; c=4.0;
    index = 1;
  }
  SECTION( "double root" ) {
    a=2.2; b=5.1; c=2.5;
    index = 2;
  }
  quadratic coefficients =
     make_tuple(a,b,c);
  auto result =
     compute_roots(coefficients);
  REQUIRE( result.index()==index );
}
```

Optional

19. Optional results

The most elegant solution to 'a number or an error' is to have a single quantity that you can query whether it's valid.

```
#include <optional>

optional<float> MaybeRoot(float x) {
  if (x<0)
    return {};
  else
    return sqrt(x);
};
  /* ... */
  for ( auto x : {2.f,-2.f} )
    if ( auto root = MaybeRoot(x) ; root.has_value() )
      cout << "Root is " << root.value() << '\n';
    else
      cout << "could not take root of " << x << '\n';
```

Exercise 7

Write a function *first_factor* that optionally returns the smallest factor of a given input.

```
auto factor = first_factor(number);
if (factor.has_value())
  cout << "Found factor: " << factor.value() << '\n';
```

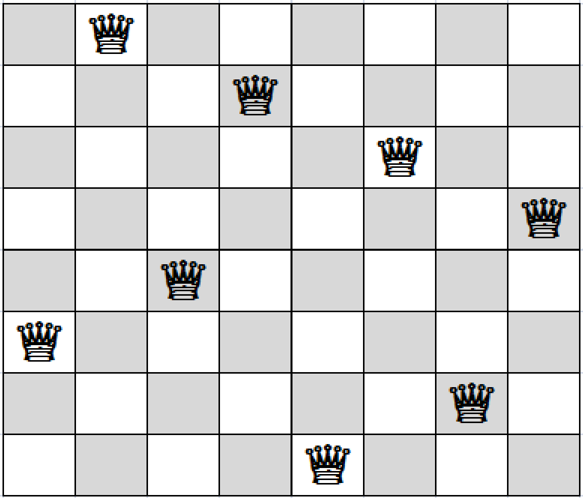
20. Any

If you want a variant that can be anything,
use `std::any`.

Eight queens problem

21. Classic problem

Can you put 8 queens on a board so that they can't hit each other?



22. Statement

- Put eight pieces on an 8×8 board, no two pieces on the same square; so that
- no two pieces are on the same row,
- no two pieces are on the same column, and
- no two pieces are on the same diagonal.

23. Not good solution

A systematic solution would run:

1. put a piece anywhere in the first row;
2. for each choice in the first row, try all positions in the second row;
3. for all choices in the first two rows, try all positions in the third row;
4. when you have a piece in all eight rows, evaluate the board to see if it satisfies the condition.

Better: abort search early.

Exercise 8: Board class

Class *board*:

```
ChessBoard(int n);
```

Method to keep track how far we are:

```
int next_row_to_be_filled()
```

Test:

```
TEST_CASE( "empty board","[1]" ) {
  constexpr int n=10;
  ChessBoard empty(n);
  REQUIRE( empty.next_row_to_be_filled()==0 );
}
```

Exercise 9: Place one queen

Method to place the next queen,
without testing for feasibility:

```
void place_next_queen_at_column(int i);
```

This test should catch incorrect indexing:

```
REQUIRE_THROWS( empty.place_next_queen_at_column(-1) );
REQUIRE_THROWS( empty.place_next_queen_at_column(n) );
REQUIRE_NOTHROW( empty.place_next_queen_at_column(0) );
REQUIRE( empty.next_row_to_be_filled()==1 );
```

Without this test, would you be able to cheat?

Exercise 10: Test if we're still good

Feasibility test:

```
bool feasible()
```

Some simple cases:
(add to previous test)

```
ChessBoard empty(n);
REQUIRE( empty.feasible() );

ChessBoard one = empty;
one.place_next_queen_at_column(0);
REQUIRE( one.next_row_to_be_filled()==1 );
REQUIRE( one.feasible() );
```

Exercise 11: Test collisions

```
ChessBoard collide = one;
// place a queen in a `colliding' location
collide.place_next_queen_at_column(0);
// and test that this is not feasible
REQUIRE( not collide.feasible() );
```

Exercise 12: Test a full board

Construct full solution

```
ChessBoard( int n,vector<int> cols );
ChessBoard( vector<int> cols );
```

Test:

```
ChessBoard five( {0,3,1,4,2} );
REQUIRE( five.feasible() );
```

Exercise 13: Exhaustive testing

This should now work:

```
// loop over all possibilities first queen
auto firstcol = GENERATE_COPY( range(1,n) );
ChessBoard place_one = empty;
REQUIRE_NOTHROW( place_one.place_next_queen_at_column(firstcol)
    );
REQUIRE( place_one.feasible() );

// loop over all possbilities second queen
auto secondcol = GENERATE_COPY( range(1,n) );
ChessBoard place_two = place_one;
REQUIRE_NOTHROW( place_two.place_next_queen_at_column(secondcol)
    );
if (secondcol<firstcol-1 or secondcol>firstcol+1) {
  REQUIRE( place_two.feasible() );
} else {
  REQUIRE( not place_two.feasible() );
}
```

Exercise 14: Place if possible

You need to write a recursive function:

```
optional<ChessBoard> place_queens()
```

- place the next queen.
- if stuck, return 'nope'.
- if feasible, recurse.

```
class board {
  /* stuff */
  optional<board> place_queens() const {
    /* stuff */
    board next(*this);
    /* stuff */
    return next;
  };
```

Exercise 15: Test last step

Test *place_queens* on a board that is almost complete:

```
ChessBoard almost( 4, {1,3,0} );
auto solution = almost.place_queens();
REQUIRE( solution.has_value() );
REQUIRE( solution->filled() );
```

Note the new constructor! (Can you write a unit test for it?)

Exercise 16: Sanity tests

```
TEST_CASE( "no 2x2 solutions","[8]" ) {
  ChessBoard two(2);
  auto solution = two.place_queens();
  REQUIRE( not solution.has_value() );
}

TEST_CASE( "no 3x3 solutions","[9]" ) {
  ChessBoard three(3);
  auto solution = three.place_queens();
  REQUIRE( not solution.has_value() );
}
```

Exercise 17: O

ptional: can you do timing the solution time as function of the size of the board?