

Standard Template Library

Victor Eijkhout, Susan Lindsey

Fall 2021

last formatted: October 12, 2021

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:

```
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

[rand] nonrandom:

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

[rand] truerandom:

Three ints: 15, 98,
70.

Time

8. 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

9. 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);
```

Tuples; Union-like stuff

10. C++11 style tuples

```
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.

11. Function returning tuple

Return type deduction:

```
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)};  
};
```


12. 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 << endl;  
else  
    cout << "Sorry, " << x  
          << " is negative" << endl;
```

Output

[stl] tuple:

Root of 2 is 1.41421
Sorry, -2 is negative

This is known as structured binding.

13. 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 << endl;  
    else  
        cout << "could not take root of " << x << endl;
```

other solution: tuples

14. Tuple solution

```
tuple<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 << endl;  
    else  
        cout << "could not take root of " << x << endl;
```

Variants

15. Variant

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

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

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

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 [a,b,c] = coefficients;  
    auto result = compute_roots(  
        coefficients);
```

Output

[union] quadratic:

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

16. 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

```
auto 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( std::abs( evaluate(coefficients,r) )<1.e-12 );
```

Exercise 5

Write a function that returns the two roots as a `std::pair`:

```
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.) );  
REQUIRE( evaluate(coefficients,r2)==Catch::Approx(0.) );
```

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

17. Optional results (C++17)

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

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

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() << endl;
//answersnippet optfactortest
/* ... */
else
    cout << "Prime number\n";

return 0;
}
```

18. Any

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