Standard Template Library

Victor Eijkhout, Susan Lindsey

Fall 2023

last formatted: February 6, 2024

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

```
// default seed
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);
```

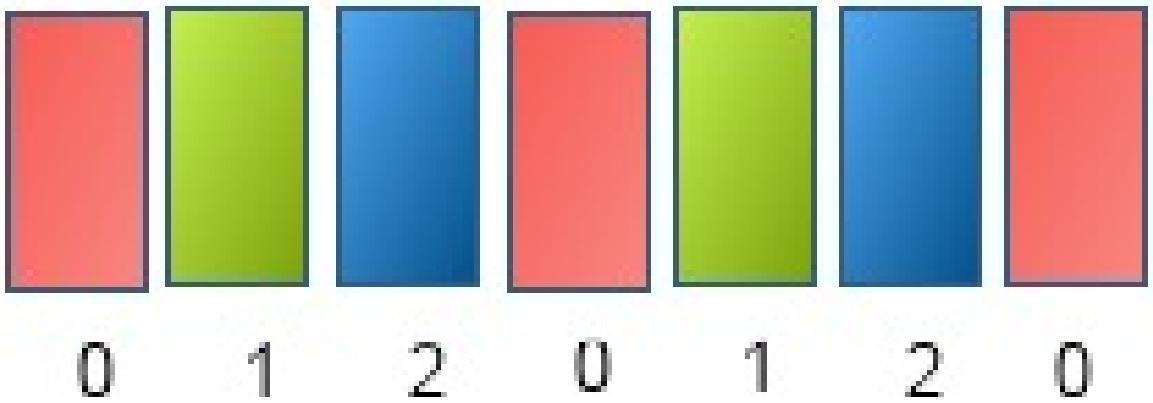
- Do not use the old C-style `random`!

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

Poisson distributed integers:
chance of k occurrences, if m is the average number
(or $1/m$ the probability)

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

7. Global engine

Good approach: random generator static in the function.

Code:

```
1 // rand/static.cpp
2 int realrandom_int(int max) {
3   static
       std::default_random_engine
       static_engine;
4   std::uniform_int_distribution<>
       ints(1,max);
5   return ints(static_engine);
6 };
```

Output:

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

A single instance is ever created.

8. Generator in a class

Note the use of `static`:

```
// rand/randname.cpp
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_percentage = 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:

```
1 // complex/veccomplex.cpp
2 vector< complex<double> >
3   vec1(N, 1.+2.5i );
4 auto vec2( vec1 );
5   /* ... */
6 for ( int i=0; i<vec1.size(); ++i )
      {
7   vec2[i] = vec1[i] * ( 1.+1.i );
8 }
9   /* ... */
10 auto sum = accumulate
11   ( vec2.begin(),vec2.end(),
12     complex<double>(0.) );
13 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. Returning tuple with type deduction

Return type deduction:

```
1  // stl/tuple.cpp
2  auto maybe_root1(float x) {
3    if (x<0)
4      return make_tuple
5        <bool,float>(false,-1);
6    else
7      return make_tuple
8        <bool,float>
9          (true,sqrt(x));
10 };
```

Alternative:

```
1  // stl/tuple.cpp
2  tuple<bool,float>
3      maybe_root2(float x) {
4    if (x<0)
5      return {false,-1};
6    else
7      return {true,sqrt(x)};
8  };
```

Note: use *pair* for *tuple* of two.

14. Catching a returned tuple

The calling code is particularly elegant:

Code:

```
1 // stl/tuple.cpp
2 auto [succeed,y] = maybe_root2(x);
3 if (succeed)
4   cout << "Root of " << x
5        << " is " << y << '\n';
6 else
7   cout << "Sorry, " << x
8        << " 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:

```
// union/optroot.cpp
bool RootOrError(float &x) {
  if (x<0)
    return false;
  else
    x = std::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

```
// union/optroot.cpp
#include <tuple>
using std::tuple, std::pair;
    /* ... */
pair<bool,float> RootAndValid(float x) {
  if (x<0)
    return {false,x};
  else
    return {true,std::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. Variant methods

```
1 // union/intdoublestring.cpp
2 variant<int,double,string> union_ids;
```

Get the index of what the variant contains:

```
1 // union/intdoublestring.cpp
2 union_ids = 3.5;
3 switch ( union_ids.index() ) {
4 case 1 :
5   cout << "Double case: " << std::get<double>(union_ids) << '\n';
6 }
```

```
1 // union/intdoublestring.cpp
2 union_ids = "Hello world";
3 if ( auto union_int = get_if<int>(&union_ids) ; union_int )
4   cout << "Int: " << *union_int << '\n';
5 else if ( auto union_string = get_if<string>(&union_ids) ; union_string
      )
6   cout << "String: " << *union_string << '\n';
```

(Takes pointer to variant, returns pointer to value)

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:

```
1 // union/quadratic.cpp
2   for ( auto coefficients :
3           { quadratic{.a=2.0,
      .b=1.5, .c=2.5},
4             quadratic{.a=1.0,
      .b=4.0, .c=4.0},
5             quadratic{.a=2.2,
      .b=5.1, .c=2.5}
6           } ) {
7     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:

```
1 // union/quadtest.cpp
2 TEST_CASE( "discriminant" ) {
3   quadratic one{0., 2.5, 0.};
4   REQUIRE( discriminant( one ) ==Catch::Approx(6.25) );
5   quadratic two{1., 0., 1.5};
6   REQUIRE( discriminant( two ) ==Catch::Approx(-6.) );
7   quadratic three{.1, .1, .1*.5};
8   REQUIRE( discriminant( three ) ==Catch::Approx(-.01) );
9 }
```

Exercise 3

Write a function

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

that passes the test

```
1 // union/quadtest.cpp
2 quadratic coefficients{a,b,c};
3 d = discriminant( coefficients );
4 z = discriminant_zero( coefficients );
5 INFO( a << "," << b << "," << c << " d=" << d );
6 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

```
1 // union/quadtest.cpp
2 auto r = simple_root(coefficients);
3 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:

```
1 // union/quadtest.cpp
2 quadratic coefficients{a,b,c};
3 auto [r1,r2] = double_root(coefficients);
4 auto
5   e1 = evaluate(coefficients,r1),
6   e2 = evaluate(coefficients,r2);
7 REQUIRE( evaluate(coefficients,r1)==Catch::Approx(0.).margin(1.e-14) );
8 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:

```
1  // union/quadtest.cpp
2  TEST_CASE( "full test" ) {
3    double a,b,c; int index;
4    SECTION( "no root" ) {
5      a=2.0; b=1.5; c=2.5;
6      index = 0;
7    }
8    SECTION( "single root" ) {
9      a=1.0; b=4.0; c=4.0;
10     index = 1;
11   }
12   SECTION( "double root" ) {
13     a=2.2; b=5.1; c=2.5;
14     index = 2;
15   }
16   quadratic
       coefficients{.a=a,.b=b,.c=c};
17   auto result =
       compute_roots(coefficients);
18   REQUIRE( result.index()==index );
19 }
```

Optional

Optional

19. Result or error

Dealing with computations that can fail:

```
bool MaybeSqrt( float &x ) {
  if ( x>=0 ) {
    x = std::sqrt(x); return true;
  } else return false;
}
```

Inelegant. Better solution:

```
optional<float> MaybeSqrt( float x ) { /* .... */ }
```

'result or no-such-thing

20. 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>
using std::optional;
```

```
1  // union/optroot.cpp
2  optional<float> MaybeRoot(float x) {
3    if (x<0)
4      return {};
5    else
6      return std::sqrt(x);
7  };
8      /* ... */
9    for ( auto x : {2.f,-2.f} )
10     if ( auto root = MaybeRoot(x) ; root.has_value() )
11       cout << "Root is " << root.value() << '\n';
12     else
13       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.

```
// primes/optfactor.cpp
auto factor = first_factor(number);
if (factor.has_value())
  cout << "Found factor: " << factor.value() << '\n';
else
else
  cout << "Prime number\n";
```

21. Mistake

Trying to take the `value` for something that doesn't have one leads to a `bad_optional_access` exception:

Code:

```
1 // union/optional.cpp
2 optional<float> maybe_number = {};
3 try {
4   cout << maybe_number.value() <<
       '\n';
5 } catch (std::bad_optional_access) {
6   cout << "failed to get value\n";
7 }
```

Output:

```
failed to get value
```

22. Expected

Expect double, return info string if not:

```
std::expected<double,string>
      square_root( double x ) {
  auto result = sqrt(x);
  if (x<0)
  return
    std::unexpected("negative");
  else if
    (x<limits<double>::min())
  return
    std::unexpected("underflow");
  else return result;
}
```

```
auto root = square_root(x);
if (x)
cout << "Root=" <<
    root.value() << '\n';
else if (root.error()==/* et
    cetera */ )
/* handle the problem */
```

Variant

23. Square root with variant

```
// union/optroot.cpp
#include <variant>
using std::variant,
    std::get_if;
    /* ... */
variant<bool,float>
    RootVariant(float x) {
  if (x<0)
    return false;
  else
    return std::sqrt(x);
};
```

```
// union/optroot.cpp
for ( auto x : {2.f,-2.f} ) {
  auto okroot = RootVariant(x);
  auto root =
    get_if<float>(&okroot);
  if ( root )
    cout << "Root is " <<
    *root << '\n';
  auto nope =
    get_if<bool>(&okroot);
  if (nope)
    cout << "could not take
    root of " << x << '\n';
}
```

24. More variant examples

Illustrating the usage:

```
1 // union/intdoublestring.cpp
2 variant<int,double,string> union_ids;
```

We can use the `index` function to see what variant is used (0,1,2 in this case) and get the value accordingly:

```
1 // union/intdoublestring.cpp
2 union_ids = 3.5;
3 switch ( union_ids.index() ) {
4 case 1 :
5   cout << "Double case: " << std::get<double>(union_ids) << '\n';
6 }
```

25. Variant methods

```
1 // union/intdoublestring.cpp
2 variant<int,double,string> union_ids;
```

Get the index of what the variant contains:

```
1 // union/intdoublestring.cpp
2 union_ids = 3.5;
3 switch ( union_ids.index() ) {
4 case 1 :
5   cout << "Double case: " << std::get<double>(union_ids) << '\n';
6 }
```

```
1 // union/intdoublestring.cpp
2 union_ids = "Hello world";
3 if ( auto union_int = get_if<int>(&union_ids) ; union_int )
4   cout << "Int: " << *union_int << '\n';
5 else if ( auto union_string = get_if<string>(&union_ids) ; union_string
      )
6   cout << "String: " << *union_string << '\n';
```

(Takes pointer to variant, returns pointer to value)

Exercise 8

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:

```
1 // union/quadratic.cpp
2   for ( auto coefficients :
3           { quadratic{.a=2.0,
      .b=1.5, .c=2.5},
4             quadratic{.a=1.0,
      .b=4.0, .c=4.0},
5             quadratic{.a=2.2,
      .b=5.1, .c=2.5}
6           } ) {
7     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
```

26. Implementation of quadratic polynomia

We represent the polynomial

$$ax^2 + bx + c$$

as

```
// union/quadlib.hpp
struct quadratic {
  double a,b,c;
};
```

Exercise 9

Write a function

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

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

```
1 // union/quadtest.cpp
2 TEST_CASE( "discriminant" ) {
3   quadratic one{0., 2.5, 0.};
4   REQUIRE( discriminant( one ) ==Catch::Approx(6.25) );
5   quadratic two{1., 0., 1.5};
6   REQUIRE( discriminant( two ) ==Catch::Approx(-6.) );
7   quadratic three{.1, .1, .1*.5};
8   REQUIRE( discriminant( three ) ==Catch::Approx(-.01) );
9 }
```

Exercise 10

Write a function

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

that passes the test

```
1 // union/quadtest.cpp
2 quadratic coefficients{a,b,c};
3 d = discriminant( coefficients );
4 z = discriminant_zero( coefficients );
5 INFO( a << "," << b << "," << c << " d=" << d );
6 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 11

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

```
1 // union/quadtest.cpp
2 auto r = simple_root(coefficients);
3 REQUIRE( evaluate(coefficients,r)==Catch::Approx(0.).margin(1.e-14) );
```

Exercise 12

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

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

Test:

```
1 // union/quadtest.cpp
2 quadratic coefficients{a,b,c};
3 auto [r1,r2] = double_root(coefficients);
4 auto
5   e1 = evaluate(coefficients,r1),
6   e2 = evaluate(coefficients,r2);
7 REQUIRE( evaluate(coefficients,r1)==Catch::Approx(0.).margin(1.e-14) );
8 REQUIRE( evaluate(coefficients,r2)==Catch::Approx(0.).margin(1.e-14) );
```

Exercise 13

Write a function

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

Test:

```
1  // union/quadtest.cpp
2  TEST_CASE( "full test" ) {
3    double a,b,c; int index;
4    SECTION( "no root" ) {
5      a=2.0; b=1.5; c=2.5;
6      index = 0;
7    }
8    SECTION( "single root" ) {
9      a=1.0; b=4.0; c=4.0;
10     index = 1;
11   }
12   SECTION( "double root" ) {
13     a=2.2; b=5.1; c=2.5;
14     index = 2;
15   }
16   quadratic
        coefficients{.a=a,.b=b,.c=c};
17   auto result =
        compute_roots(coefficients);
18   REQUIRE( result.index()==index );
19 }
```

Exercise 14

Instead of a `bool`, return a *`monostate`*.

27. Any

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