Union-like constructs

Victor Eijkhout, Susan Lindsey

Fall 2025

last formatted: November 5, 2025

Tuples

1. C++11 style tuples

Use `get` to unpack a tuple:

```
1 #include <tuple>
2
3 std::tuple<int,double,char> id = \
4     std::make_tuple<int,double,char>( 3, 5.12, 'f' );
5     // or:
6     std::make_tuple( 3, 5.12, 'f' );
7 double result = std::get<1>(id);
8 std::get<0>(id) += 1;
9
10 // also:
11 std::pair<int,char> ic = make_pair( 24, 'd' );
```

Annoyance: all that 'get'ting.

2. 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.

3. 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:

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

This is known as structured binding.

4. Returning two things

simple solution:

```
1 // union/optroot.cpp
2 bool RootOrError(float &x) {
3   if (x<0)
4     return false;
5   else
6     x = std::sqrt(x);
7   return true;
8 };
```

other solution: tuples

5. Tuples

Pack up multiple types in one variable:

```
1 // union/tuple.cpp
2 // pack:
3 tuple<int,string> intstring { 13,"fourteen" };
4 // unpack:
5 auto [i,s] = intstring;
6 println( "Int: {}, String: {}",i,s );
```

'structured binding'

6. Tuple solution

```
1  // union/optroot.cpp
2  #include <tuple>
3  using std::tuple, std::pair;
4      /* ... */
5  pair<bool,float> RootAndValid(float x) {
6    if (x<0)
7      return {false,x};
8    else
9      return {true,std::sqrt(x)};
10 };
11     /* ... */
12   for ( auto x : {2.f,-2.f} )
13     if ( auto [ok,root] = RootAndValid(x) ; ok )
14       cout << "Root is "
15            << root << '\n';
16     else
17       cout
18         << "could not take root of "
19         << x << '\n';
```

Variants

7. 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:

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

8. Problem setup

Represent the polynomial

$$ax^2 + bx + c$$

as

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

Unpack:

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

`assert` something here?

Exercise 2

Write a function

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

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

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

Exercise 3

Write a function

```
1 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:

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

```
1 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:

```
// union/quadtest.cpp
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{.a=a,.b=b
     ,.c=c};
  auto result = compute_roots(
     coefficients);
  REQUIRE( result.index()==index );
}
```

Optional

9. 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;
```

```
// union/optroot.cpp
optional<float> MaybeRoot(float x) {
  if (x<0)
    return {};
  else
    return std::sqrt(x);
};
```

Exercise 7

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

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

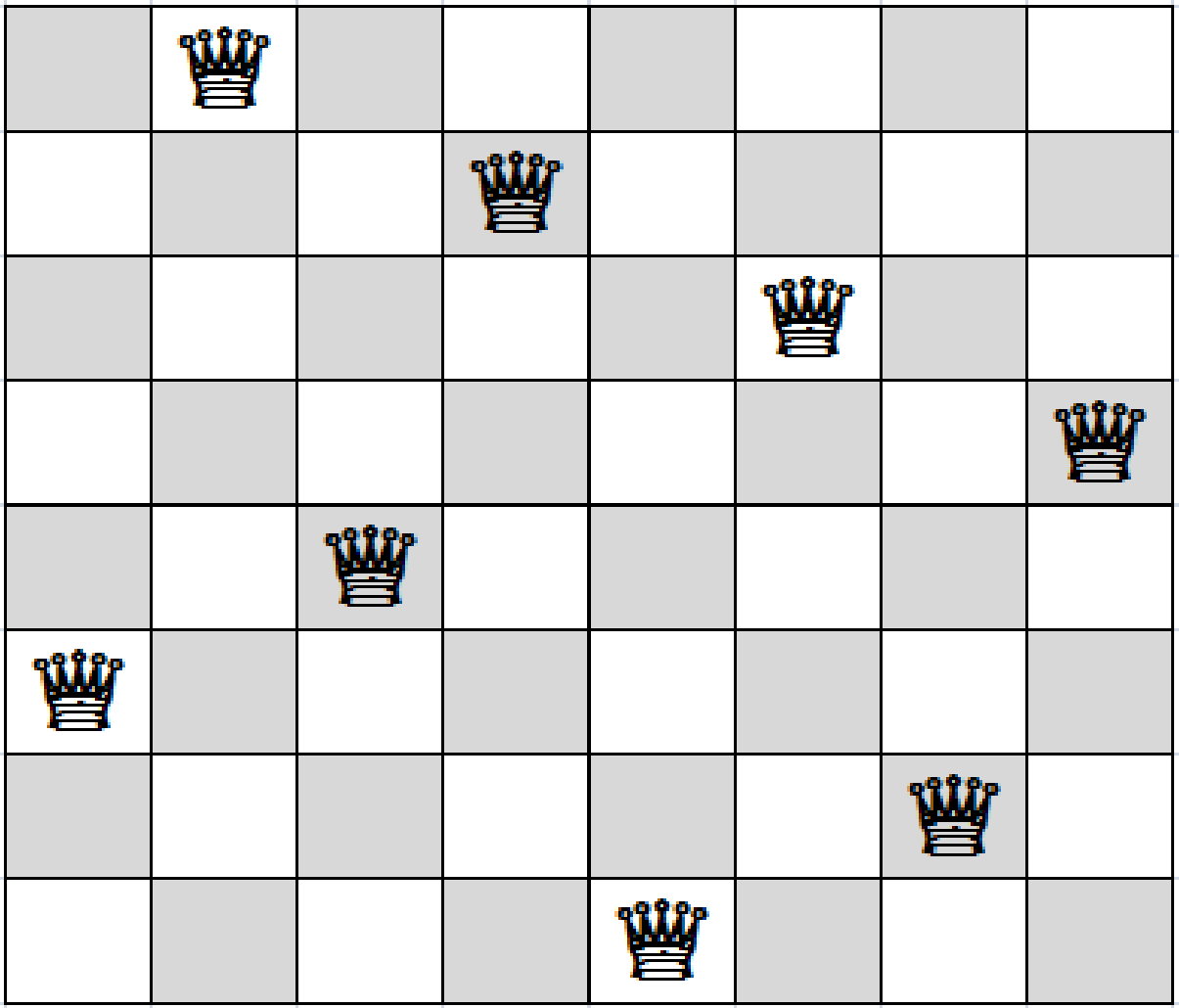
10. Any

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

Eight queens problem

11. Classic problem

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



12. 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.

13. 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*:

```
1 // queens/queens.hpp
2 ChessBoard(int n);
```

Method to keep track how far we are:

```
1 // queens/queens.hpp
2 int next_row_to_be_filled()
```

Test:

```
1 // queens/queentest.cpp
2 TEST_CASE( "empty board","[1]" ) {
3   constexpr int n=10;
4   ChessBoard empty(n);
5   REQUIRE( empty.next_row_to_be_filled()==0 );
6 }
```

Exercise 9: Place one queen

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

```
1 // queens/queens.hpp
2 void place_next_queen_at_column(int i);
```

This test should catch incorrect indexing:

```
1 // queens/queentest.cpp
2 INFO( "Illegal placement throws" )
3 REQUIRE_THROWS( empty.place_next_queen_at_column(-1) );
4 REQUIRE_THROWS( empty.place_next_queen_at_column(n) );
5 INFO( "Correct placement succeeds" );
6 REQUIRE_NOTHROW( empty.place_next_queen_at_column(0) );
7 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:

```
1 // queens/queens.hpp
2 bool feasible()
```

Some simple cases:
(add to previous test)

```
1 // queens/queentest.cpp
2 ChessBoard empty(n);
3 REQUIRE( empty.feasible() );
```

```
1 // queens/queentest.cpp
2 ChessBoard one = empty;
3 one.place_next_queen_at_column(0);
4 REQUIRE( one.next_row_to_be_filled()==1 );
5 REQUIRE( one.feasible() );
```

Exercise 11: Test collisions

```
1 // queens/queentest.cpp
2 ChessBoard collide = one;
3 // place a queen in a 'colliding' location
4 collide.place_next_queen_at_column(0);
5 // and test that this is not feasible
6 REQUIRE( not collide.feasible() );
```

Exercise 12: Test a full board

Construct full solution

```
1 // queens/queens.hpp
2 ChessBoard( int n,vector<int> cols );
3 ChessBoard( vector<int> cols );
```

Test:

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

Exercise 13: Exhaustive testing

This should now work:

```
1 // queens/queentest.cpp
2 // loop over all possibilities first queen
3 auto firstcol = GENERATE_COPY( range(1,n) );
4 ChessBoard place_one = empty;
5 REQUIRE_NOTHROW( place_one.place_next_queen_at_column(firstcol) );
6 REQUIRE( place_one.feasible() );
7
8 // loop over all possbilities second queen
9 auto secondcol = GENERATE_COPY( range(1,n) );
10 ChessBoard place_two = place_one;
11 REQUIRE_NOTHROW( place_two.place_next_queen_at_column(secondcol) );
12 if (secondcol<firstcol-1 or secondcol>firstcol+1) {
13   REQUIRE( place_two.feasible() );
14 } else {
15   REQUIRE( not place_two.feasible() );
16 }
```

Exercise 14: Place if possible

You need to write a recursive function:

```
1 // queens/queens.hpp
2 optional<ChessBoard> place_queens()
```

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

```
1 class board {
2   /* stuff */
3   optional<board> place_queens() const {
4     /* stuff */
5     board next(*this);
6     /* stuff */
7     return next;
8   };
```

Exercise 15: Test last step

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

```
1 // queens/queentest.cpp
2 ChessBoard almost( 4, {1,3,0} );
3 auto solution = almost.place_queens();
4 REQUIRE( solution.has_value() );
5 REQUIRE( solution->filled() );
```

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

Exercise 16: Sanity tests

```
1 // queens/queentest.cpp
2 TEST_CASE( "no 2x2 solutions","[8]" ) {
3   ChessBoard two(2);
4   auto solution = two.place_queens();
5   REQUIRE( not solution.has_value() );
6 }
```

```
1 // queens/queentest.cpp
2 TEST_CASE( "no 3x3 solutions","[9]" ) {
3   ChessBoard three(3);
4   auto solution = three.place_queens();
5   REQUIRE( not solution.has_value() );
6 }
```

Exercise 17: O

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