Test-Driven Development (TDD)

Victor Eijkhout, Susan Lindsey

Fall 2025

`last formatted: November 4, 2025`

1. Dijkstra quote

> *Today a usual technique is to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. (cue laughter)*

Still ...

Intro to testing

2. Types of testing

- *Unit tests* that test a small part of a program by itself;
- *System tests* test the correct behavior of the whole software system; and
- *Regression tests* establish that the behavior of a program has not changed by adding or changing aspects of it.

3. Unit testing

- Every part of a program should be testable
- ⇒ good idea to have a function for each bit of functionality
- Positive tests: show that code works when it should
- Negative tests: show that the code fails when it should

4. Unit testing

- Every part of a program should be testable
- Do not write the tests after the program:
  write tests while you develop the program.
- Test-driven development:
  1. design functionality
  2. write test
  3. write code that makes the test work

5. Principles of TDD

Develop code and tests hand-in-hand:

- Both the whole code and its parts should always be testable.
- When extending the code, make only the smallest change that allows for testing.
- With every change, test before and after.
- Assure correctness before adding new features.

6. Unit testing frameworks

Testing is important, so there is much software to assist you.

Popular choice with C++ programmers: Catch2
`https://github.com/catchorg`

7. Compiling

```
icpc -o tdd tdd.cxx \
    -I${TACC_CATCH2_INC} -L${TACC_CATCH2_LIB} \
    -lCatch2Main -lCatch2
```

- Path to include and library files:

```
-I${TACC_CATCH2_INC} -L${TACC_CATCH2_LIB}
# or:
$( pkg-config --cflags catch2 )
```

- Libraries:

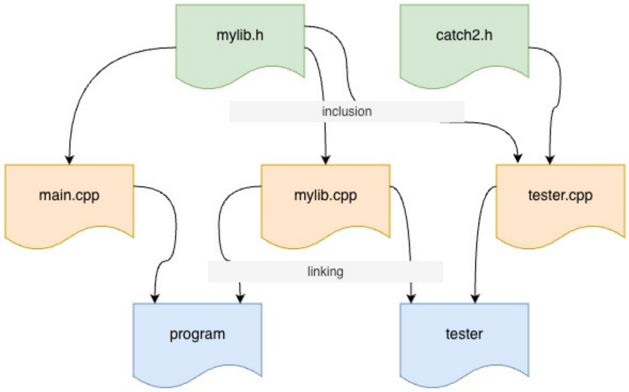
```
-lCatch2Main -lCatch2
# or:
$( pkg-config --libs catch2-with-main )
```

8. CMake

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( CATCH2 REQUIRED catch2-with-main )
target_include_directories(
        myprogram PUBLIC
        ${CATCH2_INCLUDE_DIRS}
        )
target_link_directories(
        myprogram PUBLIC
        ${CATCH2_LIBRARY_DIRS}
        )
target_link_libraries(
        myprogram PUBLIC
        ${CATCH2_LIBRARIES}
        )
```

9. Realistic setup

- All program functionality in a 'library' file
- Main program really short
- Tester file with only tests.
- (Tester also needs the `catch2` stuff included)



Exercise 1: File structure

Make three files:

1. Include file with the functions.
2. Main program that uses the functions.
3. Tester main file, contents to be determined.

10. Toy example

Function and tester:

```
// catch/positive1.cpp
#define CATCH_CONFIG_MAIN
#include "catch2/catch_all.hpp"

double f(int n) { return n*n+1; }

TEST_CASE( "test that f works once" ) {
  int n{-3};
  REQUIRE( f(n)>0 );
}
TEST_CASE( "test that f likely returns positive" ) {
  for ( auto n : {-5,-3,-1,0,1,2,3,4,5} )
    REQUIRE( f(n)>0 );
}
```

(accept the `define` and `include` as magic for now)

11. Correctness through ‘require’ clause

Tests go in `tester.cpp`:

```
TEST_CASE( "test that f always returns positive" ) {
  for (int n=0; n<1000; n++)
    REQUIRE( f(n)>0 );
}
```

- `TEST_CASE` acts like independent main program.
  can have multiple cases in a tester file
- `REQUIRE` is like `assert` but more sophisticated

Exercise 2: Safe sqrt 1

You are going to write a function `safe_sqrt`.
(Find a template in the `sessions/nov04` directory.)

Write one or two tests that confirm the correctness of your function.

12. Failure

What if your code is wrong?

Code:

```
// catch/positive3.cpp
// Wrong
double f(int n) { return n*n-2; }

TEST_CASE( "test that f returns
      positive" ) {
  for ( auto n :
      {-5,-3,-1,0,1,2,3,4,5} ) {
    INFO( "testing: " << n );
    REQUIRE( f(n)>0 );
  }
}
```

Output:

```
positive3.cpp:24:
      ↪FAILED:
  REQUIRE( f(n)>0 )
with expansion:
  -1.0 > 0
with message:
  testing: -1
```

13. Tests

Boolean:

```
REQUIRE( some_test(some_input) );
REQUIRE( not some_test(other_input) );
```

Integer:

```
REQUIRE( integer_function(1)==3 );
REQUIRE( integer_function(1)!=0 );
```

Boolean expressions need to be parenthesized:

```
REQUIRE( ( x>0 and x<1 ) );
```

14. Generating inputs

```
// catch/positive2.cpp
TEST_CASE( "test that f returns positive" ) {
  int n = GENERATE( 1,2,3,4,5 );
  REQUIRE( f(n)>0 );
}
```

```
// catch/positive2.cpp
TEST_CASE( "test that f always returns positive" ) {
  int n = GENERATE(take(100, random(-100, 100)));
  REQUIRE( f(n)>0 );
}
```

```
// catch/positive2.cpp
TEST_CASE( "test that f always returns real positive" ) {
  auto n = GENERATE(take(100, random(-100., 100.)));
  cout << n << '\n';
  REQUIRE( f(n)>0 );
}
```

Exercise 3: Safe sqrt 2

Extend your test by generating a range of inputs.

15. Numeric tests

Floating point numbers are hardly ever exact.
Do approximate tests:

```
REQUIRE( real_function(1.5)==Catch::Approx(3.0) );
REQUIRE( real_function(1)!=Catch::Approx(1.0) );
REQUIRE( zero_find()==Catch::Approx(0.).margin(1.e-8) );
```

Exercise 4: Safe sqrt 3

Extend your test by testing that your square root function delivers approximately the right result.

16. Diagnostic information for failing tests

`INFO`: print out information at a failing test

```
TEST_CASE( "test that f always returns positive" ) {
  for (int n=0; n<1000; n++)
    INFO( "iteration: " << n );
    REQUIRE( f(n)>0 );
}
```

17. Test for exceptions

Suppose function `g(n)`

- succeeds for input $n > 0$
- fails for input $n \leq 0$:
  throws exception

```
TEST_CASE( "test that g only works for positive" ) {
  for (int n=-100; n<+100; n++)
    if (n<=0)
      REQUIRE_THROWS( g(n) );
    else
      REQUIRE_NOTHROW( g(n) );
}
```

Exercise 5: Safe sqrt 4

Extend your function: a negative input should cause an exception.

Test for this.

18. Tests with code in common

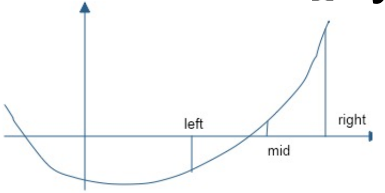
Use SECTION if tests have intro/outtro in common:

```
TEST_CASE( "commonalities" ) {
  // common setup:
  double x,y,z;
  REQUIRE_NOTHROW( y = f(x) );
  // two independent tests:
  SECTION( "g function" ) {
    REQUIRE_NOTHROW( z = g(y) );
  }
  SECTION( "h function" ) {
    REQUIRE_NOTHROW( z = h(y) );
  }
  // common followup
  REQUIRE( z>x );
}
```

(sometimes called setup/teardown)

TDD example: Bisection

19. Root finding by bisection



- Start with bounds where the function has opposite signs.

$$x_- < x_+, \qquad f(x_-) \cdot f(x_+) < 0,$$

- Find the mid point;
- Adjust either left or right bound.

20. Coefficient handling

$$f(x) = c_0 x^d + c_1 x^{d-1} \cdots + c_{d-1} x^1 + c_d$$

We implement this by storing the coefficients in a `vector<double>`.
Proper:

```
// root/testzeroarray.cpp
TEST_CASE( "coefficients represent polynomial" "[1]") {
  vector<double> coefficients = { 1.5, 0., -3 };
  REQUIRE( coefficients.size()>0 );
  REQUIRE( coefficients.front()!=0. );
}
```

Exercise 6: One test for properness

For polynomial coefficients to give a well-defined polynomial, the zero-th coefficient needs to be non-zero:

```
// bisect/zeroclasstest.cpp
TEST_CASE( "proper test","[2]" ) {
  vector<double> coefficients{3., 2.5, 2.1};
  REQUIRE_NOTHROW( polynomial(coefficients) );

  coefficients.at(0) = 0.;
  REQUIRE_THROWS( polynomial(coefficients) );
}
```

Write a constructor that accepts the coefficients, and throws an exception if the above condition is violated.

21. Handy shortcut

Are you getting tired of typing `vector<double>`?
put

```
// root/findzerolib.hpp
using polynomial = vector<double>;
```

somewhere high in your file.

22. Test on polynomials evaluation

Next we need to evaluate polynomials.

Equality testing on floating point is dangerous:

use *Catch::Approx(sb)*

```
// root/testzeroclass.cpp
polynomial second( {2,0,1} );
// correct interpretation: 2x^2 + 1
REQUIRE( second.is_proper() );
REQUIRE( second.evaluate_at(2) == Catch::Approx(9) );
// wrong interpretation: 1x^2 + 2
REQUIRE( second.evaluate_at(2) != Catch::Approx(6) );
```

Exercise 7: Implementation

Write a function `evaluate_at` which computes

$$y \leftarrow f(x).$$

and confirm that it passes the above tests.

```
double evaluate_at( polynomial coefficients,double x);
```

For bonus points, look up Horner's rule and implement it.

23. Odd degree polynomials only

With odd degree you can always find bounds x_-, x_+.
For this exercise we reject even degree polynomials:

```
// root/findzeroarray.cpp
if ( not is_odd(coefficients) ) {
  cout << "This program only works for odd-degree polynomials\n";
  exit(1);
}
```

This test will be used later;
first we need to implement it.

Exercise 8: Odd degree testing

Implement the *is_odd* test.

Gain confidence by unit testing:

```
// root/testzeroarray.cpp
polynomial second{2,0,1}; // 2x^2 + 1
REQUIRE( not is_odd(second) );
polynomial third{3,2,0,1}; // 3x^3 + 2x^2 + 1
REQUIRE( is_odd(third) );
```

24. Finding initial bounds

We need a function `find_initial_bounds` which computes x_-, x_+ such that

$$f(x_-) < 0 < f(x_+) \quad \text{or} \quad f(x_+) < 0 < f(x_-)$$

(can you write that more compactly?)

```
void find_initial_bounds
  ( polynomial coefficients,double &left,double &right);
```

Since we reject even degree polynomials,
throw an exception for those.

Exercise 9: Test for initial bounds

Unit test:

```
// root/testzeroarray.cpp
right = left+1;
polynomial second{2,0,1}; // 2x^2 + 1
REQUIRE_THROWS( find_initial_bounds(second,left,right) );
polynomial third{3,2,0,1}; // 3x^3 + 2x^2 + 1
REQUIRE_NOTHROW( find_initial_bounds(third,left,right) );
REQUIRE( left<right );
```

Can you add a unit test on the left/right values?

25. Move the bounds closer

Root finding iteratively moves the initial bounds closer together:

```
move_bounds_closer(coefficients,left,right);
```

- on input, `left<right`, and
- on output the same must hold.

Design a test for this function;
implement this function.

26. Putting it all together

Ultimately we need a top level function

```
double find_zero( polynomial coefficients,double prec );
```

- reject even degree polynomials
- set initial bounds
- move bounds closer until close enough: $|f(y)| < \mathtt{prec}$.

Exercise 10: Put it all together

Make this call work:

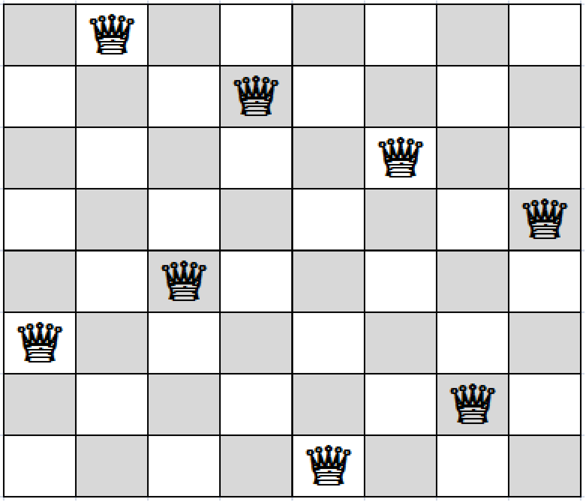
```
// root/findzeroarray.cpp
auto zero = find_zero( coefficients, 1.e-8 );
cout << "Found root " << zero
     << " with value " << evaluate_at(coefficients,zero) << '\n';
```

Design unit tests, including on the precision attained, and make sure your code passes them.

Eight queens problem by TDD (using objects)

27. Problem statement

Can you place eight queens on a chess board so that no pair threatens each other?

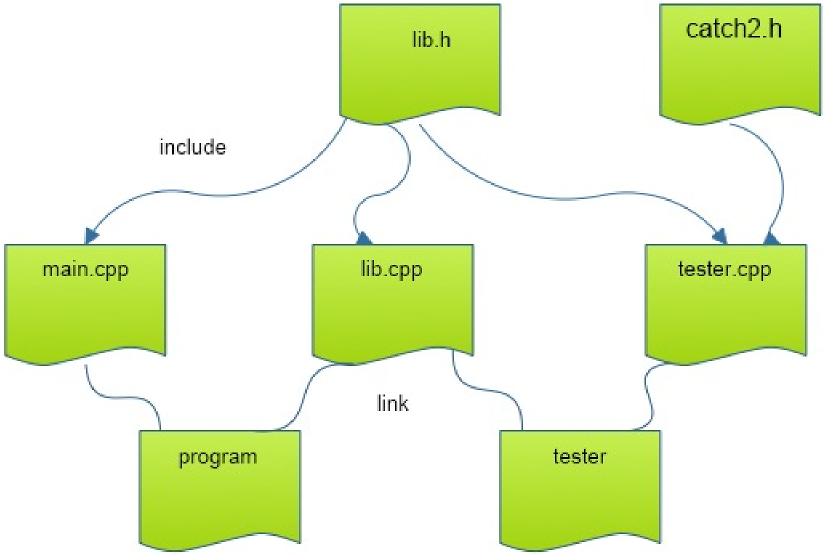


28. Sort of test-driven development

You will solve the ‘eight queens’ problem by

- designing tests for the functionality
- then implementing it

29. File structure



30. Basic object design

Object constructor of an empty board:

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

Test how far we are:

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

First test:

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

Exercise 11: Board object

Start writing the *board* class, and make it pass the above test.

Exercise 12: Board method

Write a method for placing a queen on the next row,

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

and make it pass this test (put this in a TEST_CASE):

```
// queens/queentest.cpp
INFO( "Illegal placement throws" )
REQUIRE_THROWS( empty.place_next_queen_at_column(-1) );
REQUIRE_THROWS( empty.place_next_queen_at_column(n) );
INFO( "Correct placement succeeds" );
REQUIRE_NOTHROW( empty.place_next_queen_at_column(0) );
REQUIRE( empty.next_row_to_be_filled()==1 );
```

Exercise 13: Test for collisions

Write a method that tests if a board is collision-free:

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

This test has to work for simple cases to begin with. You can add these lines to the above tests:

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

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

```
// queens/queentest.cpp
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 14: Test full solutions

Make a second constructor to ‘create’ solutions:

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

Now we test small solutions:

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

Exercise 15: No more delay: the hard stuff!

Write a function that takes a partial board, and places the next queen:

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

Test that the last step works:

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

Alternative to using `optional`:

```
bool place_queen( const board& current, board &next );
// true if possible, false is not
```

Exercise 16: Test that you can find solutions

Test that there are no 3×3 solutions:

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

but 4×4 solutions do exist:

```
// queens/queentest.cpp
TEST_CASE( "there are 4x4 solutions","[10]" ) {
  ChessBoard four(4);
  auto solution = four.place_queens();
  REQUIRE( solution.has_value() );
}
```