Error handling and testing

Victor Eijkhout, Susan Lindsey

COE 322 Fall 2021

1. Programming and correctness

Find your favorite example of costly programming mistakes . . .

What to do about it?

- Never make mistakes.
- Prove that your program is correct.
- Test your program before deploying it.
- Handle errors as they occur.

Error handling

2. Assertions to catch logic errors

Sanity check on things 'that you just know are true':

```
#include <cassert>
...
assert( bool expression )
```

Example:

```
x = sin(2.81);
y = x*x;
z = y * (1-y);
assert( z>=0. and z<=1. );
```

3. Using assertions

Check on valid input parameters:

```
#include <cassert>

// this function requires x<y
// it computes something positive
float f(x,y) {
  assert( x<y );
  return /* some result */;
}
```

Check on valid results:

```
float positive_outcome = f(x,y);
assert( positive_outcome>0 );
```

4. Example

```
int collatz_next( int current ) {
  assert( current>0 );
  int next{-1};
  if (current%2==0) {
    next = current/2;
    assert(next<current);
  } else {
    next = 3*current+1;
    assert(next>current);
  }
  return next;
}
```

5. Use assertions during development

Assertions are disabled by

```
#define NDEBUG
```

before the include.

You can pass this as compiler flag:
`icpc -DNDEBUG yourprog.cxx`

6. Exceptions

Not every error is fatal:

$$\text{Exception} \equiv \begin{cases} \text{'this should not happen'} \\ \text{but we can handle it} \end{cases}$$

1. recover from the problem
2. graceful exit

7. Exceptions

Have you seen the following?

Code:

```
vector<float> x(5);
x.at(5) = 3.14;
```

Output

```
[except] boundthrow:

libc++abi.dylib: terminating with
    uncaught exception of type std
    ::out_of_range: vector
make[2]: *** [run_boundthrow]
    Abort trap: 6
```

The Standard Template Library (STL) can generate many exceptions.

- You can let your program crash, and start debugging
- You can try to catch and handle them yourself.

8. Exception structure

Code with problem:

```
if ( /* some problem */ )
  throw(5);
  /* or: throw("error"); */
```

```
try {
  /* code that can go wrong */
} catch (...) { // literally
    three dots!
  /* code to deal with the
    problem */
}
```

9. Exceptions

Assume a routine only works for certain values, and you want to generate an error if called with an inappropriate value.

```
double compute_root(double x) {
  if (x<0) throw(1);
  return sqrt(x);
}
int main() {
  try {
    y = compute_root(x);
  } catch (...) {
    /* handle error */
    cout << "Root failed, using default\n";
    y = 0;
  }
```

See book for more details.

Unit testing and test-driven development (TDD)

10. 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 . . .

11. 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.

12. Test-driven development

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.

13. Unit testing frameworks

Testing is important, so there is much software that assists you.

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

14. Toy example

Function and tester:

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

#define CATCH_CONFIG_MAIN
#include "catch2/catch_all.hpp"

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

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

15. Compiling toy example

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

- Files:

  ```
  icpc -o tdd tdd.cxx
  ```

- Path to include and library files:

  ```
  -I${TACC_CATCH2_INC} -L${TACC_CATCH2_LIB}
  ```

- Libraries:

  ```
  -lCatch2Main -lCatch2
  ```

Exercise 1: Extend the toy example

1. Write a function

   ```
   double f(int n) { /* .... */ }
   ```

   with values in the range $(0, 1)$.
2. Write a unit test for this.

You can base this off the file `tdd.cxx` in the repository

16. Slightly realistic example

Example: we use a function that

- only works for positive inputs;
- returns input $+1$.

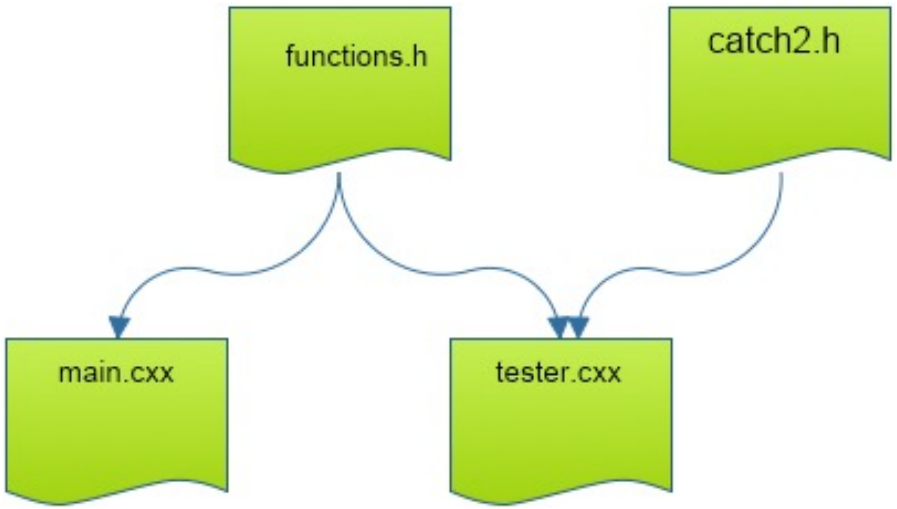
Program that uses this:

```
#include "functions.h"
int main() {
  for ( int i=10; i>-1; i-- )
    cout << "One more than the positive number "
         << i << " is "
         << increment_positive_only(i)
         << "\n";
```

Note the include file!

17. File structure

- All program functionality in a 'library' file
- Main program really short
- Second main with only tests.



18. Function to be developed

We know the structure:

```
int increment_positive_only( int i ) {
  // this function returns one more than the inpuut
  // input has to be positive, error otherwise
  /* ... */
}
```

function body to be developed.

19. Functionality testing

File `tester.cxx`:

Same include file for the functionality;
the testing framework creates its own main.

```
#include "functions.h"

#define CATCH_CONFIG_MAIN
#include "catch2/catch_all.hpp"

TEST_CASE( "test the increment function" ) {
  /* ... */
}
```

20. Compiling the tester

One-line solution:

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

21. Compile and link options

Variables for a Makefile:

```
INCLUDES = -I${TACC_CATCH2_INC}
EXTRALIBS = -L${TACC_CATCH2_LIB} -lCatch2Main -lCatch2
```

Exercise 2: 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.

22. Correctness through 'require' clause

Tests go in `tester.cxx`:

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

- *TEST_CASE* acts like independent program.
- REQUIRE is like assert but more sophisticated
- Can contain (multiple) tests for correctness.

23. Output for failing tests

Run the tester:

```
-------------------------------------------------------------------
test the increment function
-------------------------------------------------------------------
test.cxx:25
...............................................................

test.cxx:29: FAILED:
  REQUIRE( increment_positive_only(i)==i+1 )
with expansion:
  1 == 2

===================================================================
test cases: 1 | 1 failed
assertions: 1 | 1 failed
```

24. 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( "function fails for " << n );
    REQUIRE( f(n)>0 );
}
```

Exercise 3: Positive tests

Continue with the example of slide 20:
add a positive `TEST_CASE`

```
for (int i=1; i<10; i++)
  REQUIRE( increment_positive_only(i)==i+1 );
```

Make the function satisfy this test.

25. Test for exceptions

Supppose 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 4: Negative tests

Make sure your function throws an exception at illegal inputs:

```
for (int i=0; i>-10; i--)
  REQUIRE_THROWS( increment_positive_only(i) );
```

26. Tests with code in common

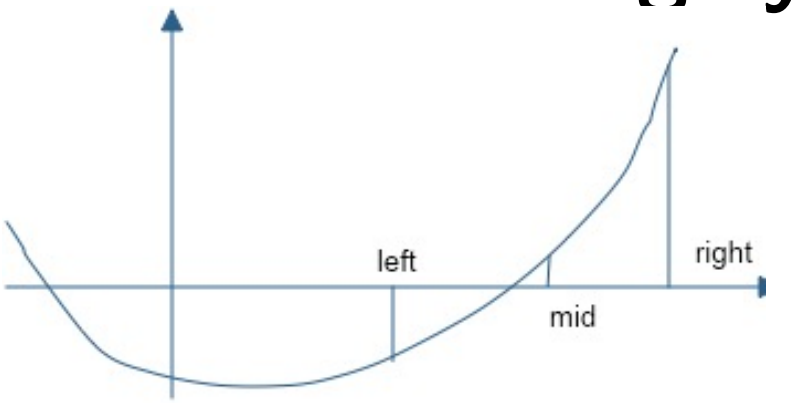
Use SECTION if tests have intro/outtro in common:

```
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

27. 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.

28. Coefficient handling

$$f(x) = c_d x^d + \cdots + c_1 x^1 + c_0$$

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

```
TEST_CASE( "coefficients are polynomial","[1]" ) {
  auto coefficients = set_coefficients();
  REQUIRE( coefficients.size()>0 );
  REQUIRE( coefficients.front()!=0. );
}
```

Exercise 5: Proper polynomials

Write a routine *set_coefficients* that constructs a vector of coefficients:

```
vector<double> coefficients = set_coefficients();
```

and make it satisfy the above conditions.

At first write a hard-coded set of coefficients, then try reading them from the command line.

Exercise 6: One test for properness

Write a test *proper_polynomial* and write unit tests for it, both passing and failing.

29. Test on polynomials evaluation

```
// correct interpretation: 2x^2 + 1
vector<double> second{2,0,1};
REQUIRE( proper_polynomial(second) );
REQUIRE( evaluate_at(second,2) == Catch::Approx(9) );
// wrong interpretation: 1x^2 + 2
REQUIRE( evaluate_at(second,2) != Catch::Approx(6) );
```

Exercise 7: Implementation

Write a function *evaluate_at* which computes

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

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

Exercise 8: Odd degree polynomials only

With odd degree you can always find bounds x_-, x_+.
Reject even degree polynomials:

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

Gain confidence by unit testing:

```
vector<double> second{2,0,1}; // 2x^2 + 1
REQUIRE( not is_odd(second) );
vector<double> third{3,2,0,1}; // 3x^3 + 2x^2 + 1
REQUIRE( is_odd(third) );
```

Exercise 9: Find bounds

Write a function *find_outer* 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?)
Unit test:

```
right = left+1;
vector<double> second{2,0,1}; // 2x^2 + 1
REQUIRE_THROWS( find_outer(second,left,right) );
vector<double> third{3,2,0,1}; // 3x^3 + 2x^2 + 1
REQUIRE_NOTHROW( find_outer(third,left,right) );
REQUIRE( left<right );
```

How would you test the function values?

Exercise 10: Put it all together

Make this call work:

```
auto zero = find_zero(coefficients,left,right);
cout << "Found root " << zero
     << " with value " << evaluate_at(coefficients,zero) << "\n";
```

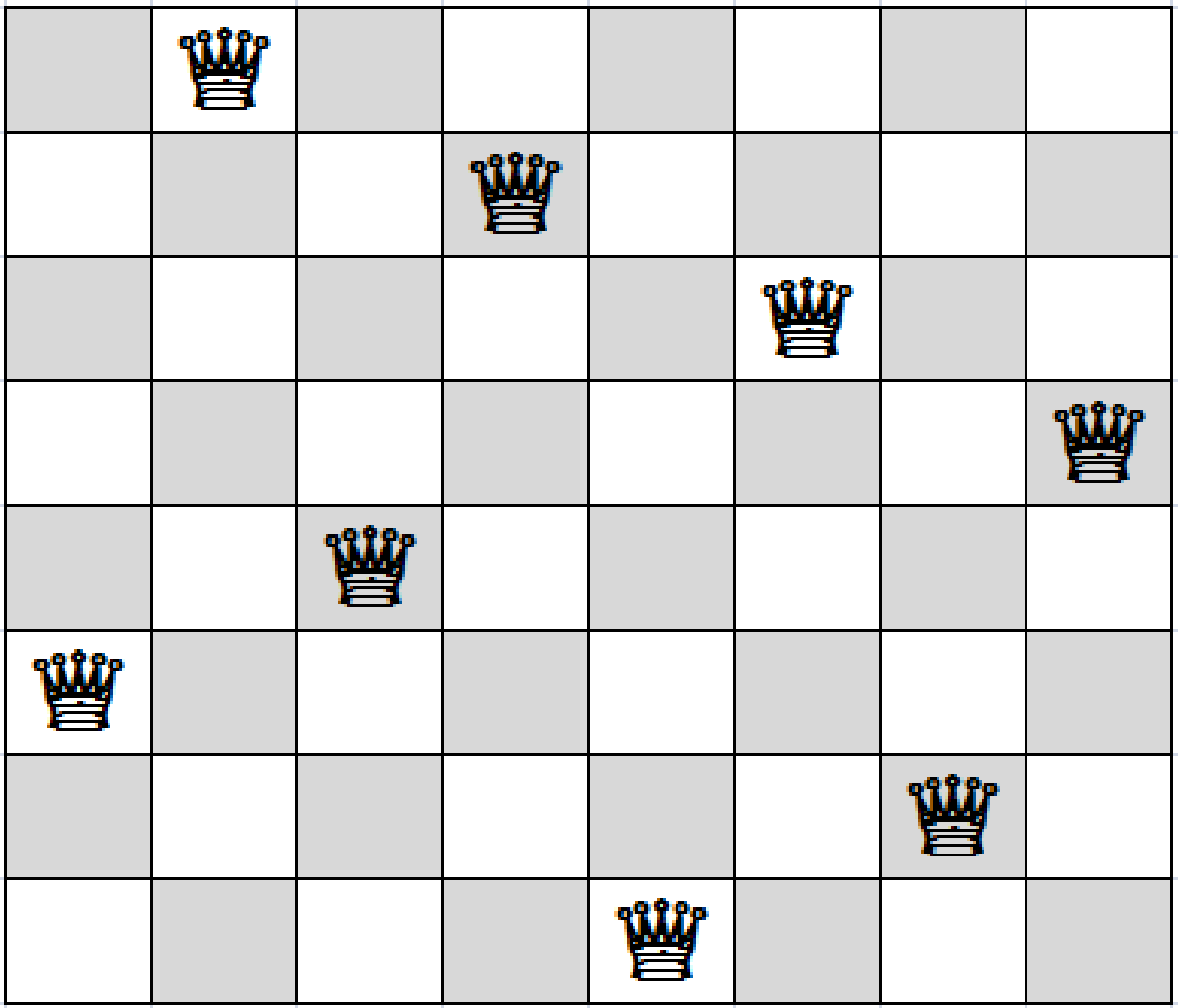
Add an optional precision argument to the root finding function.

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

Eight queens problem

30. Problem statement

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

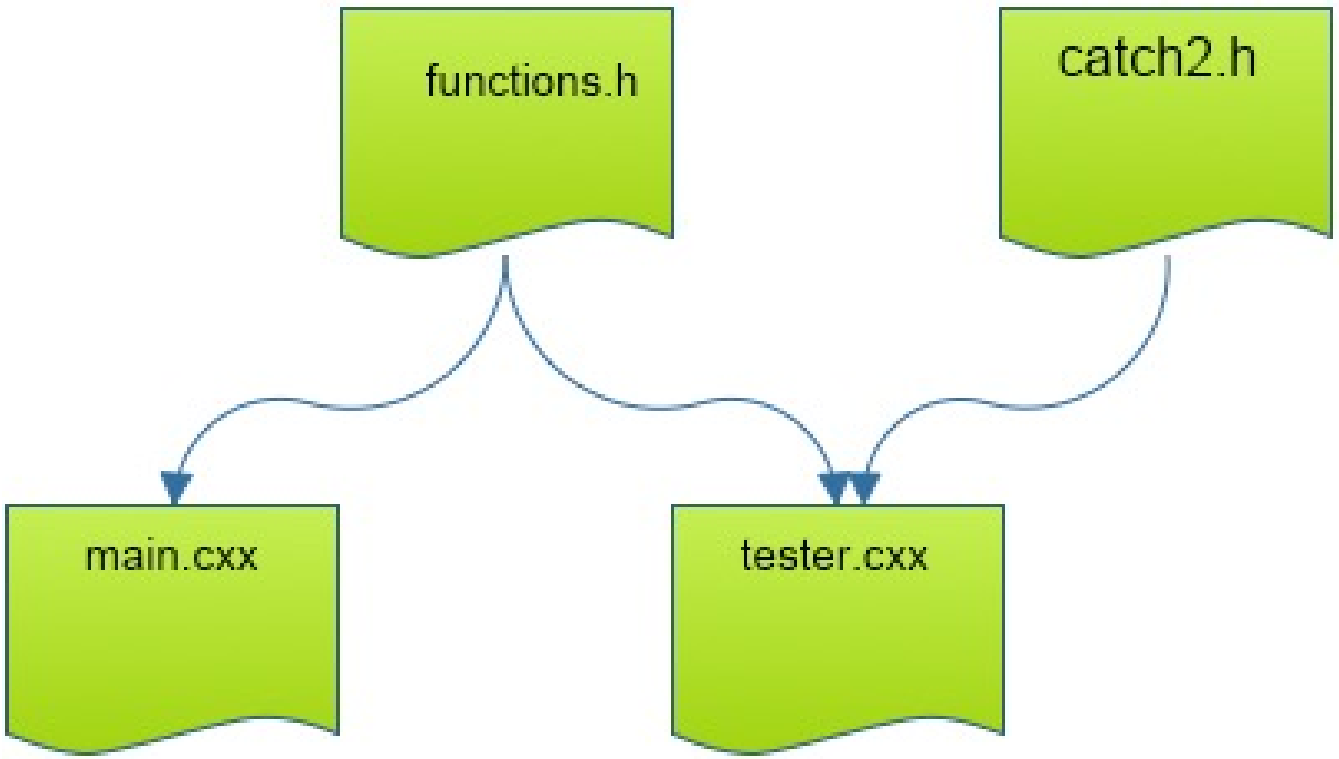


31. Sort of test-driven development

You will solve the 'eight queens' problem by

- designing tests for the functionality
- then implementing it

32. File structure



33. Basic object design

Object constructor of an empty board:

```
board(int n);
```

Test how far we are:

```
int next_row_to_be_filled() const;
```

First test:

```
TEST_CASE( "empty board" ) {
  constexpr int n=10;
  board 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,

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

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

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

Exercise 13: Test for collisions

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

```
bool feasible() const;
```

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

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

REQUIRE( one.feasible() );

auto collide(one);
collide.place_next_queen_at_column(0);
REQUIRE( not collide.feasible() );
```

Exercise 14: Test full solutions

Make a second constructor to 'create' solutions:

```
board( vector<int> cols );
```

Now we test small solutions:

```
board 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:

```
optional<board> place_queen(const board& current);
```

Test that the last step works:

```
board almost( {1,3,0,board::magic::empty} );
auto solution = place_queen(almost);
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:

```
TEST_CASE( "no 3x3 solutions" ) {
  board three(3);
  auto solution = place_queen(three);
  REQUIRE( not solution.has_value() );
}
```

but 4×4 solutions do exist:

```
TEST_CASE( "there are 4x4 solutions" ) {
  board four(4);
  auto solution = place_queen(four);
  REQUIRE( solution.has_value() );
}
```