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. Use assertions during development

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

Assertions are disabled by

```
#define NDEBUG
```

before the include.

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

3. Using assertions

```
#include <cassert>

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

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

4. Exceptions

Have you seen the following?

Code:

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

Output
[except] boundthrow:

```
libc++abi.dylib: terminating
make[2]: *** [run_boundthrow]
```

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.

5. 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 (...) { // literally three dots!
    /* handle error */
  }
```

See book for more details.

Unit testing and test-driven development (TDD)

6. 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 . . .

7. 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.

8. 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.

9. Unittesting frameworks

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

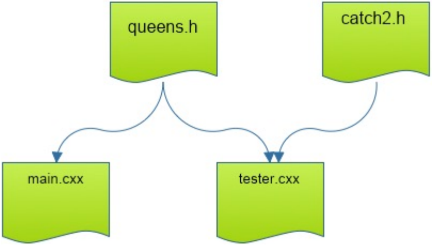
Popular choice with C++ programmers: Catch2
`https://github.com/catchorg`

10. Compile and link options

```
INCLUDES = -I${TACC_CATCH2_INC}
EXTRALIBS = -L${TACC_CATCH2_LIB} -lCatch2 -lCatch2main
```

11. File structure

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



12. Unittest driver file

The framework supplies its own main:

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

#include "library_functions.h"
/*
   here follow the unit tests
*/
```

13. Correctness through ‘require’ clause

- *TEST_CASE* acts like independent program.
- Can contain (multiple) tests for correctness.

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

14. Test for exceptions

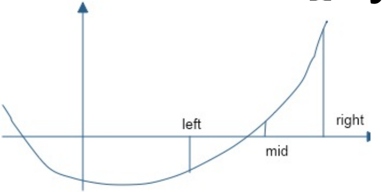
```
TEST_CASE( "test that g only works for positive" ) {
  for (int n=-100; n<+100; n++)
    if (n<=0)
      REQUIRE_THROWS( g(n) );
    else
      REQUIRE_NOTHROW( g(n) );
}
```

15. Setup and teardown

```
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 );
}
```

TDD example: Bisection

16. 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.

Exercise 1: Polynomial construction

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

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

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

Exercise 2: 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:

```
TEST_CASE( "polynomial degree" ) {
  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 3: Evaluation

Write an evaluation function and test it:

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

Exercise 4: 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 5: 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";
```

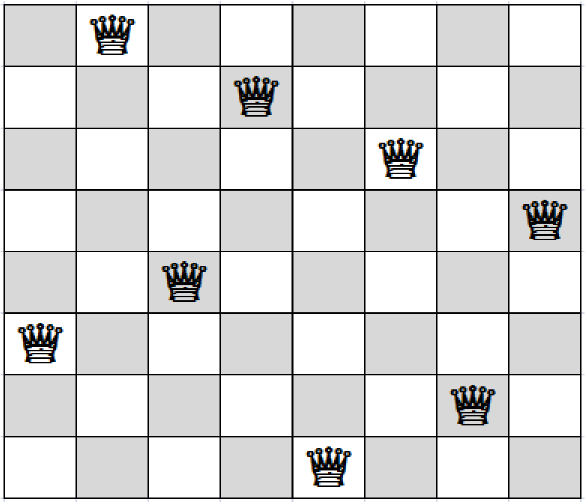
Add an optional precision argument to the root finding function.

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

Eight queens problem

17. Problem statement

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

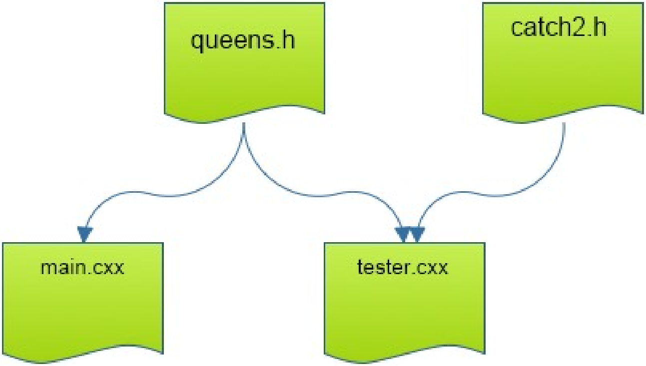


18. Sort of test-driven development

You will solve the 'eight queens' problem by

- designing tests for the functionality
- then implementing it

19. File structure



20. 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 6: Board object

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

Exercise 7: 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 8: 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 9: 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 10: 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 11: 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() );
}
```