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’: 

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

Example:

```c
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:

```c
#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:

```c
float positive_outcome = f(x,y);
assert( positive_outcome>0 );
```
4. Example

```c
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:

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:**

```cpp
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
```

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:

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

```java
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.

```c++
double compute_root(double x) {
    if (x<0) throw(1);
    return sqrt(x);
}
```

```c++
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. 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
13. 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
14. 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.
15. Unit testing frameworks

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

Popular choice with C++ programmers: Catch2
https://github.com/catchorg
16. Toy example

Function and tester:

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

```c
#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)
17. 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
   
   ```c
   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*
18. 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)
19. Slightly realistic example

Example: we use a function that

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

Program that uses this:

```cpp
#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!
20. Function to be developed

We know the structure:

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

function body to be developed.
21. 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" ) {
  /* ... */
}
```
22. Compiling the tester at TACC

One-line solution:

icpc -o tester test_main.cxx \
   -I${TACC_CATCH2_INC} -L${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.
23. Correctness through ‘require’ clause

Tests go in tester.cxx:

```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 program.
- `REQUIRE` is like `assert` but more sophisticated
- Can contain (multiple) tests for correctness.
24. Tests

labels: catch-approx Boolean:

\begin{verbatim}
REQUIRE( some_test(some_input) );
REQUIRE( not some_test(other_input) );
\end{verbatim}

Integer:

\begin{verbatim}
REQUIRE( integer_function(1)==3 );
REQUIRE( integer_function(1)!==0 );
\end{verbatim}

Beware floating point:

\begin{verbatim}
REQUIRE( real_function(1.5)==Catch::Approx(3.0) );
REQUIRE( real_function(1)!==Catch::Approx(1.0) );
\end{verbatim}

In general exact tests don't work.
25. Output for failing tests
Run the tester:

--------------------------------
test the increment function
class Test {
public:
  void testIncrement() {
    REQUIRE(increment_positive_only(1) == 2);
  }
};
--------------------------------

test.cxx:25
................................
test.cxx:29: FAILED:
  REQUIRE(increment_positive_only(1) == 2 + 1)
with expansion:
  1 == 2

================================
test cases: 1 | 1 failed
assertions: 1 | 1 failed
26. 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 23: add a positive TEST_CASE

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

Make the function satisfy this test.
27. Test for exceptions

Suppose function g(n)

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

```cpp
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:

```python
for (int i=0; i>-10; i--)
    REQUIRE_THROWS(increment_positive_only(i));
```
**28. Tests with code in common**

Use `SECTION` if tests have intro/outtro in common:

```cpp
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
29. Root finding by bisection

- Start with bounds where the function has opposite signs.
  \[ x_- < x_+, \quad f(x_-) \cdot f(x_+) < 0, \]
- Find the mid point;
- Adjust either left or right bound.
30. 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>`.

```cpp
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:

```cpp
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 function `proper_polynomial` as described, and write unit tests for it, both passing and failing.
31. 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

\[ y \leftarrow f(x). \]

and confirm that it passes the above tests.

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:

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

Gain confidence by unit testing:

```cpp
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:

```cpp
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:

```cpp
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.
Turn it in!

• If you think your functions pass all tests, subject them to the tester:
  coe_bisection yourprogram.cc
  where ‘yourprogram.cc’ stands for the name of your source file.

• Is it reporting that your program is correct? If so, do:
  coe_bisection -s yourprogram.cc
  where the -s flag stands for ‘submit’.

• If you don’t manage to get your code working correctly, you can submit as incomplete with
  coe_bisection -i yourprogram.cc

• If you want feedback on what the tester thinks about your code do
  coe_bisection -d yourprogram.cc
  with the -d flag for ‘debug.'
Eight queens problem
32. Problem statement
Can you place eight queens on a chess board so that no pair threatens each other?
33. Sort of test-driven development

You will solve the ‘eight queens’ problem by

• designing tests for the functionality
• then implementing it
34. File structure

- functions.h
- catch2.h
- main.cxx
- tester.cxx
35. Basic object design

Object constructor of an empty board:

\[ \text{board}(\text{int } \text{n}); \]

Test how far we are:

\[ \text{int } \text{next_row_to_be_filled}() \text{ const}; \]

First test:

\[
\text{TEST\_CASE}( \text{"empty board" } ) \{ \\
\text{constexpr int } \text{n}=10; \\
\text{board empty(n);} \\
\text{REQUIRE( empty.next_row_to_be_filled()==0 ); }
\}
\]
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,

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

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

```cpp
auto one(empty);
REQUIRE_THROWS( one.place_next_queen_at_column(-1) );
REQUIRE_THROWS( one.place_next_queen_at_column(n) );
REQUIRE_NOTTHROW( 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:

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

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

```cpp
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:

\[
\text{board( } \text{vector<int> cols } );
\]

Now we test small solutions:

\[
\text{board five( } \{0,3,1,4,2\} );
\text{ REQUIRE( five.feasible() );}
\]
Exercise 15: No more delay: the hard stuff!

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

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

Test that the last step works:

```cpp
board almost( {1,3,0,board::magic::empty} );
auto solution = place_queen(almost);
REQUIRE( solution.has_value() );
REQUIRE( solution->filled() );
```

Alternative to using `optional`:

```cpp
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 $3 \times 3$ solutions:

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

but $4 \times 4$ solutions do exist:

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