Chapter 68

Unit testing and Test-Driven Development

In an ideal world, you would prove your program correct, but in practice that is not always feasible, or at least: not done. Most of the time programmers establish the correctness of their code by testing it.

Yes, there is a quote by *Edsger Dijkstra* that goes:

> 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)

but that doesn't mean that you can't at least gain some confidence in your code by testing it.

68.1 Types of tests

Testing code is an art, even more than writing the code to begin with. That doesn't mean you can't be systematic about it. First of all, we distinguish between some basic types of test:

- *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.

In this section we will talk about unit testing.

A program that is written in a sufficiently modular way allows for its components to be tested without having to wait for an all-or-nothing test of the whole program. Thus, testing and program design are aligned in their interests. In fact, writing a program with the thought in mind that it needs to be testable can lead to cleaner, more modular code.

In an extreme form of this you would write your code by Test-Driven Development (TDD), where code development and testing go hand-in-hand. The basic principles can be stated as follows:

- 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.

In a strict interpretation, you would even for each part of the program first write the test that it would satisfy, and then the actual code.

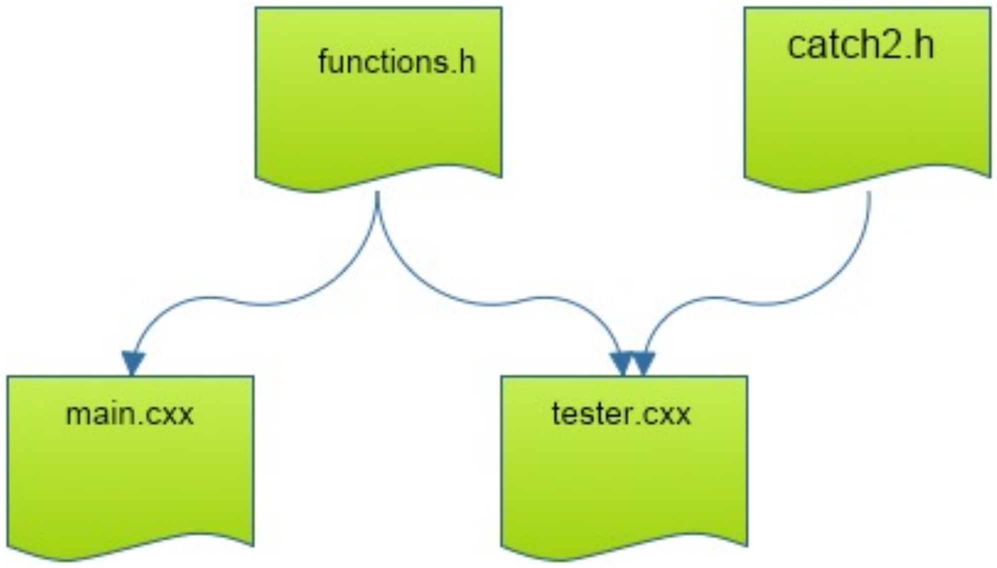

Figure 68.1: File structure for unit tests

68.2 Unit testing frameworks

There are several 'frameworks' that help you with unit testing. In the remainder of this chapter we will use *Catch2*, which is one of the most used ones in C++.

You can find the code at `https://github.com/catchorg`.

68.2.1 File structure

Let's assume you have a file structure witth

- a very short main program, and
- a library file that has all functions use by the main.

In order to test the functions, you supply another main, which contains only unit tests; This is illustrated in figure 68.1.

In fact, with Catch2 your main file doesn't actually have a `main` program: that is supplied by the framework. In the tester main file you only put the test cases.

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
*/
```

One imporant question is what header file to include. You can do

```
#include "catch.hpp"
```

which is the ‘header only’ mode, but that makes compilation very slow. Therefore, we will assume you have installed Catch through *Cmake*, and you include

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

Note: as of September 2021 this requires the development version of the repository, not any `2.x` release.

68.2.2 Compilation

The setup suggested above requires you to add compile and link flags to your setup. This is system-dependent.

Compiling the tester at TACC One-line solution:

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

Compile and link options at TACC Variables for a Makefile:

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

68.2.3 Test cases

A test case is a short program that is run as an independent main. In the setup suggested above, you put all your unit tests in the tester main program, that is, the file that has the

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

magic lines.

Each test case needs to have a unique name, which is printed when a test fails. You can optionally add keys to the test case that allow you to select tests from the commandline.

```
TEST_CASE( "name of this test" ) {
  // stuf
}
TEST_CASE( "name of this test","[key1][key2]" ) {
  // stuf
}
```

The body of the test case is essentially a main program, where some statements are encapsulated in test macros. The most common macro is `REQUIRE`, which is used to demand correctness of some condition.

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.

Exercise 68.1.

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`.

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
```

For failing tests you get a diagnostic message, but in the above case it would print out the offending value of `f(n)`, not the value of `n` for which it occurs. To determine this, insert `INFO` specifications, which only get print out if a test fails.

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 );
}
```

If your code throws exceptions (section 23.2.2) you can test for these.

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) );
}
```

A common occurrence in unit testing is to have multiple tests with a common setup or teardown, to use terms that you sometimes come across in unit testing. Catch2 supports this: you can make 'sections' for part in between setup and teardown.

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)

68.3 An example: the eight queens problem

Let's revisit the *eight queens* exercise of section 7.7.1, and tackle it with unit tests. The basic strategy will be that we fill consecutive rows, by indicating each time which column will be occupied by the next queen.

Using an Object-Oriented (OO) strategy, we make a class `board` with a constructor that only indicates the size of the problem:

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

We have an auxiliary function that states what the next row to be filled is:

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

This gives us our first trivial test: