## Contents

## 1 include/leap_year.hpp

```cpp
/*
 * Toy Test - Toy Unit Testing
 * Written in 2018 by Gerald Lewis <lewisgdljr@gmail.com>
 *
 * To the extent possible under law, the author(s) have dedicated all copyright
 * and related and neighboring rights to this software to the public domain
 * worldwide. This software is distributed without any warranty.
 * You should have received a copy of the CC0 Public Domain Dedication along
 * with this software. If not, see
 * <http://creativecommons.org/publicdomain/zero/1.0/>.
 */

#define INTENTIONAL_FAILURE
//># `bool is_leap_year(int year)`
//>Calculates whether the parameter, `year`, was a leap year.
//>There's a `#define` to make the test fail, `INTENTIONAL_FAILURE`.
//>This causes the function to NOT throw an exception for years < 1752,
//>which is an error because that was the year that the Gregorian calendar
//>was adopted by the British Empire. (Although it was used by other
//>European nations before that, so it's not an error in those countries...)
//>But I basically just put it in to demonstrate the THROWS() macro.
//>
bool is_leap_year( int year ) {
#ifndef INTENTIONAL_FAILURE
   if ( year < 1752 ) {
      // is the year one in which the Gregorian calendar
      // was used in the British Empire and/or USA?
      throw std::invalid_argument(
         "The Gregorian calendar wasn't used in the "
         "British Empire (and therefore the American colonies) before 1752!" );
   }
#endif // INTENTIONAL_FAILURE

   if ( ( year & 3 ) || ( !( year % 100 ) && ( year % 400 ) ) ) {
      // is the year odd or not a multiple of 4?
      // or is the year an even century but NOT a multiple of 400 years?
      return false;
   }

   return true;
}
```

## 2 include/factorial.hpp

```cpp
/*
 * Toy Test - Toy Unit Testing
 * Written in 2018 by Gerald Lewis <lewisgdljr@gmail.com>
 *
 * To the extent possible under law, the author(s) have dedicated all copyright
 * and related and neighboring rights to this software to the public domain
 * worldwide. This software is distributed without any warranty.
 * You should have received a copy of the CC0 Public Domain Dedication along
 * with this software. If not, see
 * <http://creativecommons.org/publicdomain/zero/1.0/>.
 */

//># `int factorial(int n)`
//>A simple version of the factorial function.
//>Caveat: Does not perform error checks!
//>
int factorial( int n ) {
   if ( n < 1 )
      return 1;
   return n * factorial( n - 1 );
}
```

## 3 ndoc.sh

```bash
#!/bin/bash

file_pattern="${1:-*.hpp}"
file_location_raw="${2:-include}"
file_location="${file_location_raw#./}"

extract_doc()
{
    local curr_file;
    curr_file="${1#${file_location}/}"
    echo "" >> "doc/Documentation.md"
    echo "## ${curr_file}" >> "doc/Documentation.md"
    echo "" >> "doc/Documentation.md"
    grep '//>' "${1}" | sed -e 's%.*//>[[:space:]]*%%g' -e 's/^#/###/' >> "doc/
        Documentation.md"
}

for i in $(find "${file_location}" -name "${file_pattern}"); do
    extract_doc $i;
done
```

## 4 samples/Makefile

```
.PHONY: all clean

vpath %.hpp ../include

INCLUDES = ../include
CPPFLAGS = -I"$(INCLUDES)"
CXXFLAGS = -std=c++17
CC = $(CXX)
SOURCES = $(wildcard *.cpp)
OBJECTS = $(subst .cpp,.o,$(SOURCES))
TARGETS = $(patsubst %.cpp,%,$(SOURCES))
TARGETS_WIN = $(subst .cpp,.exe,$(SOURCES))

all: $(TARGETS)

%.d: %.cpp
	$(CXX) $(CXXFLAGS) -MM $(CPPFLAGS) $< > $@.$$$$; \
	sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
	rm -f $@.$$$$

clean:
	rm $(OBJECTS) $(TARGETS) $(TARGETS_WIN) 2> /dev/null || true

include $(subst .cpp,.d,$(SOURCES))
```

## 5 test/factorial-test.hpp

```
namespace factorial_internal
{
  //># toy_test::suite factorial_suite
  //>A sample test suite. Tests the factorial function in factorial.hpp
  //>

  toy_test::suite factorial_suite{
   "Test for factorial",
   {
      {"0! == 1", [] { ASSERT( factorial( 0 ) == 1 ); }},
      {"3! == 6", [] { ASSERT( factorial( 3 ) == 6 ); }},
      {"10! == 3628800", [] { ASSERT( factorial( 10 ) == 3628800 ); }},
   }};
}
using factorial_internal::factorial_suite;
```

## 6 test/leap-year-test.hpp

```
namespace leap_year_internal
{
  //># toy_test::suite leap_year_suite
  //>A sample test suite. Tests the leap year formula function in leap_year.hpp
  //>
  toy_test::suite leap_year_suite {
   "Test for leap year formula",
   {{"odd years are not leap years", [] { ASSERT( !is_leap_year( 2001 ) ); }},

    {"even years which are not multiples of 4 are not leap years",
     [] { ASSERT( !is_leap_year( 2002 ) ); }},

    {"multiples of 4 but not 100 are leap years",
     [] { ASSERT( is_leap_year( 1996 ) ); }},

    {"multiples of 100 but not 400 are not leap years",
     [] { ASSERT( !is_leap_year( 1900 ) ); }},

    {"multiples of 400 are leap years", [] { ASSERT( is_leap_year( 2000 ) ); }},

    {"years before 1752 are not valid",
     [] { THROWS( is_leap_year( 800 ), std::exception ); }}}};

}

using leap_year_internal::leap_year_suite;
```

## 7 test/toy_test.hpp

```
/*
 * Toy Test - Toy Unit Testing
 * Written in 2018 - 2020 by Gerald Lewis <lewisgdljr@gmail.com>
 *
 * To the extent possible under law, the author(s) have dedicated all copyright
 * and related and neighboring rights to this software to the public domain
 * worldwide. This software is distributed without any warranty.
 * You should have received a copy of the CC0 Public Domain Dedication along
 * with this software. If not, see
 * <http://creativecommons.org/publicdomain/zero/1.0/>.
 */

#pragma once
#ifndef TOY_TEST_HPP_INCLUDED
#define TOY_TEST_HPP_INCLUDED

#include <functional>
#include <initializer_list>
#include <iostream>
#include <vector>

//># namespace toy_test
//>
namespace toy_test {
  //>## `class toy_test::test_case`
  //>An element of a `suite`. Contains a name and an anonymous function.
  //>Usually created anonymously, as an element of an array within a `suite`.
  //>
  struct test_case {
    const char*           name;
    std::function<void()> run;
    void                  operator()() const { run(); }
  };

  //>## `class toy_test::failure`
  //>Holds information about a `test_case`'s failure.
  //>Not usually instantiated directly, but from the failure of an `ASSERT`.
  //>
  struct failure {
    const char* expr;
    int         line;
  };

  //>## `class toy_test::suite`
  //>A container for `test_case`s. Holds a name and a `std::vector`
  //>of `test_case`s. Usually initialized using aggregate initialization.
  //>
  struct suite {
    const char*            name;
    std::vector<test_case> tests;
    //>### `bool toy_test::suite::run()`
    //>Function that executes the `test_case`s in a `suite`.
    //>It prints the results of the test run, including how many
    //>`test_case`s succeeded and failed.
    //>
    bool run() const {
      auto ok = true;
      auto count = 0;
      auto count_ok = 0;
      auto count_fail = 0;
      std::cout << "[SUITE] Running test suite: \"" << name << "\""
                << std::endl;
      for ( auto&& test : tests ) {
        try {
          ++count;
          test();
          ++count_ok;
          std::cout << "[OK] \"" << test.name << "\" passed."
                    << std::endl;
        } catch ( failure& caught ) {
          ok = false;
          ++count_fail;
          std::cout << "[FAIL] \"" << test.name << "\" failed."
                    << std::endl;
          std::cout << "Failing condition: \"" << caught.expr
                    << "\" at line: " << caught.line << std::endl;
        }
      }
      if ( ok ) {
        std::cout << "[SUITE] " << count_ok << "/" << count << " tests passed."
                  << std::endl;
      } else {
        std::cout << std::endl
                  << "[WARNING] Test failures detected in suite: \"" << name
                  << "\"" << std::endl
                  << "[WARNING] " << count_ok <<"/" << count << " tests passed and "
                  << count_fail << "/" << count << " tests failed."
                  << std::endl;
      }
      return ok;
    }
  };

  //>## `bool toy_test::run_suite(suite const& suite)`
  //>Runs the contents of a `suite`.
  //>Returns `false` if there were any failures.
  //>
  bool run_suite( suite const& suite ) {
    auto result = suite.run();
    std::cout << std::endl;
    return result;
  }

  //>## `bool toy_test::run_suites(std::initializer_list<suite const> const& suites)`
  //>Runs the contents of a set of `suite`s, given as a list-initialized
  //>parameter. Returns `false` if there were any failures in any `suite`.
  //>
  bool run_suites( std::initializer_list<suite const> const& suites ) {
    bool ok = true;
    for ( auto const& a : suites ) {
      ok &= run_suite( std::forward<suite const>( a ) );
    }
```

```cpp

    if ( ok ) {
      std::cout << "All tests passed." << std::endl
                << std::endl;
    } else {
      std::cout << "[WARNING] Test failures detected." << std::endl
                << " Check the output for details." << std::endl
                << std::endl;
    }
    return ok;
  }
  // end of namespace toy_test
}

//># `macro ASSERT(condition)`
//>Tests a condition. Fails the `test_case` if the condition is `false`.
//>Also aborts the `test_case` on failure.
//>
#define ASSERT( condition )                                             \
  void( ( condition ) ? 0                                               \
        : throw toy_test::failure(                                      \
                                  {"ASSERT(" #condition ")", __LINE__} ) )

//># `macro THROWS(expression, exception)`
//>Tests to ensure that a provided expression causes a particular exception,
//>or one of its subtypes, is thrown. Fails the `test_case` and aborts it if
//>the expected exception is not thrown.
//>
#define THROWS( expression, exception )                                 \
  try {                                                                 \
    ( expression );                                                     \
    throw toy_test::failure(                                            \
                            {"THROWS(" #expression ", " #exception ")", __LINE__} ); \
  } catch ( exception& ) {                                              \
  } catch ( ... ) {                                                     \
    throw toy_test::failure(                                            \
                            {"THROWS(" #expression ", " #exception ")", __LINE__} ); \
  }

#endif // TOY_TEST_HPP_INCLUDED
```

## 8 test/Makefile

```
.PHONY: all clean test

vpath %.hpp ../include

INCLUDES = ../include
CPPFLAGS = -I"$(INCLUDES)"
CXXFLAGS = -std=c++17 ${CFLAGS}
CC = $(CXX)
SOURCES = $(wildcard *.cpp)
OBJECTS = $(subst .cpp,.o,$(SOURCES))
TARGETS = $(patsubst %.cpp,%,$(SOURCES))
TARGETS_WIN = $(subst .cpp,.exe,$(SOURCES))

all: $(TARGETS)

%.d: %.cpp $(INCLUDES)/*.hpp
	$(CXX) $(CXXFLAGS) -MM $(CPPFLAGS) $< > $@.$$$$; \
	sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
	rm -f $@.$$$$

clean:
	rm $(OBJECTS) $(TARGETS) $(TARGETS_WIN) 2> /dev/null || true

test: $(TARGETS)
	for t in $(TARGETS) ; do ./$$t ; done

include $(subst .cpp,.d,$(SOURCES))
```

## 9 test/demo-test.hpp

```
//># `namespace demo_test_internal`
//>Just a `namespace` to wrap the test `suite` and any
//>necessary types and variables so they don't pollute the global `namespace`.
//>
namespace demo_test_internal
{
  //>## `toy_test::suite demo_test_internal::demo_suite`
  //>This is the test `suite`. It has one `test_case` that passes,
  //>and one that fails.
  //>
  auto demo_suite = toy_test::suite
    {
     "demonstration test suite",
     {
      {"passes",
       [] {
         ASSERT(true);
       }},

      {"fails",
       [] {
         ASSERT(false);
       }},


     }};
}

//># `toy_test::suite demo_suite`
//>This is a variable alias to make the test `suite` available
//>in the global `namespace`. The idea is that if you have
//>`using` statements, `typedef`s,  and variables inside the
//>`suite`'s `namespace`, they don't pollute the global `namespace`.
//>However, we can still address the `suite` itself using a global variable.
//>
using demo_test_internal::demo_suite;
```

## 10 test/test-suite.cpp

```cpp
/*
 * Toy Test - Toy Unit Testing
 * Written in 2018 by Gerald Lewis <lewisgdljr@gmail.com>
 *
 * To the extent possible under law, the author(s) have dedicated all copyright
 * and related and neighboring rights to this software to the public domain
 * worldwide. This software is distributed without any warranty.
 * You should have received a copy of the CC0 Public Domain Dedication along
 * with this software. If not, see
 * <http://creativecommons.org/publicdomain/zero/1.0/>.
 */

#include <factorial.hpp>
#include <leap_year.hpp>
#include "toy_test.hpp"
#include "leap-year-test.hpp"
#include "factorial-test.hpp"
#include "demo-test.hpp"

int main() {
  toy_test::run_suites( {leap_year_suite,
                         factorial_suite,
                         demo_suite,
    } );
}
```

## 11 Makefile

```
.PHONY: all test clean distclean samples doc

OVERKILL = -pedantic -Wall -Wcast-align -Wcast-qual -Wctor-dtor-privacy -Wdisabled-optimization -Wdouble-promotion -Wduplicated-branches -Wduplicated-cond -Werror -Wextra -Wfatal-errors -Wfloat-equal -Wformat=2 -Winit-self -Winline -Wlogical-op -Wlto-type-mismatch -Wmissing-include-dirs -Wold-style-cast -Woverloaded-virtual -Wpedantic -Wredundant-decls -Wshadow -Wshadow-local -Wsign-conversion -Wsign-promo -Wstrict-overflow=5 -Wswitch-default -Wundef -Wuseless-cast

#CFLAGS = -Wall -Werror -Wpedantic -pedantic -Wfatal-errors

CFLAGS = ${OVERKILL}

all: samples

test:
        make -C test test CFLAGS="${CFLAGS}"

samples:
        make -C samples all CFLAGS="${CFLAGS}"

clean:
        make -C test clean
        make -C samples clean

distclean: clean
        find . -name "*.d" -delete || true
        find . -name "*~" -delete || true

doc: doc/Documentation.md

doc/Documentation.md: include/*.hpp
        echo "# Documentation" > "doc/Documentation.md"
        ./ndoc.sh *.hpp .
```