Unit testing with C++

Author: Damir Ljubic
email: damirlj@yahoo.com

Introduction

Unit Testing is probably not a very popular topic among the SW developers - at least not when it comes to writing the unit tests, but let's hope it will be interesting reading material.
I've recently participated in two-days training organized by the company - about writing the unit tests conform with the ASPICE requirements, so this would be kind of a reflection on the content of the course - trying to summarize the highlights.

ASPICE

Automotive SPICE defines certain quality gates that software as the final product needs to satisfy in order to be positively assessed, and your company "white listed" along with other suppliers that can actually deliver to the OEMs - well to a certain degree, depending on the stage - level you've reached in the assessment process.

From the aspect of SW development, ASPICE defines certain levels:

- **SW Requirements Analysis (SWE.1)**
  It about the the functional (and non-functional: quality) requirements as inputs into SW Architecture, but usually seen at the level of what is labeled as SW Element

  @note SW Element has meaning of the building block of the software, that can be *unit* (the smallest building block: like functions, or class), *component* (more complex building blocks like classes or modules), and *high-level components* (the binaries - apps, developed and maintained in collaboration, within the entire team)

- **SW Architectural design (SWE.2)**
  It's about SW Element descriptions in terms of the external interfaces and dependencies with other SW Elements.
  We usually talk here about the IPC: Service-Client like interfaces between the high-level components in responsibility of different teams: as a clear separations of the concerns

- **SW Detailed Design and Unit Controls (SWE.3)**
  Here we deal with the SW Element internal structure and design (along with documentation), and the concrete implementation and testing.
  On this level, we can perform different kind of testing, from top-to-bottom
  - **Quality tests** (whether acceptance criteria are satisfied)
  - **Integration tests** (integration with other components: the IPC)
  - **Component verification** (unit tests)

Now, when we briefly described SW Engineering levels, we can talk more about the unit testing

[digression] *I've been tempted to say, now that we reached the bottom , as an allusion to the introduction punchline: that we consider Unit testing as some side dish, although it's far from that - it's a main course, with a lot of knowledge required to make it right.*

Unit testing

From the perspective of the Quality Management (ASPICE), the unit testing will be assessed through the test coverage - that is, merely through the **metrics** (like statement, branch, conditional coverage), while our goal, as developer, should be the quality of the tests - and above all: design for testability

DIP vs IOSP

Dependency Inversion Principle is one of the SOLID principles coined by Robert C. Martin, that advocates instead of having high-level code that directly depends on the low-level (platform/library dependent interface) code, we introduce a new abstraction level - as a way to decouple them, so that both depend on the newly introduced interface (contract).
This way, the high-level code knows only how to call it, without being aware of any implementation details, while low-level code needs to empower this interface with the real implementation.
This way, we can mock the low-level code: for the purposes of the writing unit tests.

This can be implemented in various ways, depending on the use-case.
One way is to use CRTP and static polymorphism: by specifying the new abstraction level via the "base" class - and then provide the derived classes specific implementation.

Integration Operation Segregation Principle is also a clean-code principle.
What it means is that we should strive in designing our code to break the complex code into small operational - independent steps (Single Responsibility Principle), and have a single integration method: that assembles all these steps, without introducing any business logic.

@note This is similar to the Template Method design pattern

This way, we have a method resilient to change: the operational steps - the business logic (implementation details) can be changed, without affecting the integration method itself.
The integration method can be changed - but only by adding a new operation.
This way - operations can be tested in isolation (unit tests), while the integration method itself - as name implies, becomes a subject of integration tests: in case that it contains the dependencies that have to be otherwise mocked (DIP).

Let's demonstrate this through an example.
Assume that we have a method that fetch the users from database, and do something meaningful with that

```
template <typename Predicate, typename Output>
void show_users_by(Predicate&& pred, Output&& out, std::string_view
dbName) {
   std::vector<User> users = fetch_users_from_db(dbName);
   auto filtered_users = filter_users(users,
std::forward<Predicate>(pred));
   show_users(std::move(filtered_users), std::forward<Output>(out));
}
```

DIP approach would be to mock the fetching the user from database - by introducing the new level of abstraction: interface.
IOSP approach would be to write the unit tests for the operational steps that don't carry any dependencies (fetching/showing), while the remaining functionality - fetching from a real database, will be covered by the integration test.

Takeaway is: always prefer IOSP over DIP in terms of design for testability.

The other mantra - TDD, or in our case **Test First** approach is another extreme in software design that requires a completely new mindset and discipline and it's out of scope of this article. But shortly, Test First approach means:

- Design test scenario for which the test fails due to missing implementation (red)
- Implement the missing feature/property: so that test passes (green)
- Refactor the ongoing implementation
- Repeat it, until the implementation is completed

Test frameworks

For C++ development, the two most popular test frameworks are gtest and catch2.

We use Catch2 as a single-header test framework developed by Phil Nash, and it's more than suitable for writing the unit tests at native side - testing usually some utility, helper classes: that can be run without any dependencies (software, and/or hardware) on local machine.
Unlike gtest framework - it doesn't support mocking scenarios, but as we seen, with clever design (IOSP) and combining different type of tests - we can have desired test coverage

In light of everything that has been written so far, let's demonstrate a certain test strategy on the simple requirement.

[Requirement] Write the function, that turns any collection into formatting string as "[a1, a2, …, an]" - for a logging purpose

[digression] *there is already std::format function - format library introduced with C++20, that satisfies this requirement, but this is not the point of this exercise*

```
#include <fmt/format.h>
#include <fmt/ranges.h>

template <typename Collection>
[[nodiscard]] auto collection_to_str(const Collection& collection)
{
   return fmt::format("[{}]", fmt::join(collection, ", "));
}
```

We can implement this requirement, using **Dispatcher Pattern**: similar to IOSP guidance, we have a single method that has a compile-time switch for invoking the proper implementation based on the collection elements type: without being aware of the implementation details.
It's also easy extendible - for handling some other special cases, if needed

```
template <class Iterator>
[[nodiscard]] auto collection_to_str(Iterator first, Iterator last)
{
   using value_type = std::decay_t<decltype(*first)>;

   // Special handling for the enum types
   if constexpr (std::is_enum_v<value_type>)
   {
       return enum_collection_to_str(first, last);
   }
   // Special handling for "byte-like" types
   else if constexpr (is_byte_type<value_type>)      {
       return byte_collection_to_str<Iterator, value_type>(first,
last);
```

```
    }
    else { return basic_collection_to_str(first, last); }
}
```

The overloaded version takes, for convenience, the collection as input argument

```
template <class Collection>
[[nodiscard]] auto collection_to_str(const Collection& collection)
{
    return collection_to_str(std::cbegin(collection),
std::cend(collection));
}
```

What about the testing strategies?

Since there shouldn't be any unpredictable outcomes for the method under test, we should be fine with an **Example-based test** (AAA - *Arrange Act Assert*): testing it as a black box, validating the outputs - comparing them with the expected results, for the limited number of inputs: as our testing set.

The advantage of this approach is that it is intuitive - easy to implement and understand, without involving any additional - test logic.
In most cases - this should be the first: proper choice for writing the unit tests.

The possible test case, using catch2, could be as follow

```
TEST_CASE("Algorithms: testing collection_to_str() method, with
std::string", "[algorithms][collection_to_str][string]")
{

    SECTION("collection_to_str_checkOutputForLimitedInputs_Ok")
    {
        using namespace std::string_literals;
        using namespace utils::algorithm;

        using StringVector = std::vector<std::string>;
        auto [input, expected] = GENERATE(table<StringVector,
std::string>(
        {
            {StringVector{}, "[]"s},
            {StringVector{","}, "[,]"s},
            {StringVector{"A"}, "[A]"},
            {StringVector{"eins", "zwei", "drei", "vier", "fünf"},
"[eins, zwei, drei, vier, fünf]"s},
```

```
            {StringVector{"eins-1", "zwei-2", "drei-3", "fünf-5"},
"[eins-1, zwei-2, drei-3, fünf-5]"s}
        }));

        CHECK(collection_to_str(input) == expected);
    }
}
```

Some notes, before we proceed.

- We can either have different test cases for the method, or within the same test case - we can have different **sections** within the same test case - that cover different aspects of the test: the inputs for which test should succeed, vs. those for which it should fail, etc. Or, in this case, for different inputs - elements type
- **Naming**: it's an important attribute of a unit test.
  We will uniformly use the **[method name]_[test scenario]_[outcome]** pattern as naming convention.
  This makes the test self-explanatory and self-documented at the same time.
- **Tagin**: this enables, during the execution - to separate the test runs, for the particular test cases with the same tagging

GENERATE (table<>) is a macro that stores the given test set as a map (actually, a tuple) of inputs with associated - expected outputs, and loop around them, applying whatever logic follows: in this case, simply checking the method output.
Quite convenient, instead defining your own map (std::unordered_map) and looping around it with for-loop

Nice. What if we have a more demanding algorithm, for which we want to write a unit test to check the various properties of this algorithm, on an arbitrary - randomly generated input set?

Obviously, we can't look at the method as black box anymore - we need to be clever enough to design the tests that cover different aspects - different properties of the algorithm.
And we need to test it on a wide range of randomly generated data: for better coverage - to easier detect: bubble up the issue

There is a name for this: **Property-based tests**

The biggest disadvantage of this approach is to have a test logic - designed to test the properties of the algorithm.
But also, to randomly generate meaningful test data.
Fortunately, there is a test framework (probably more) - **rapidcheck**, that provide different generators, and works for both - gtest, and catch2
Link: [rapidcheck/doc/generators_ref.md at master · emil-e/rapidcheck · GitHub]

For our simple algorithm - the property would be the format of the output string.

We can - for the purpose of the demonstration, try to write the property-based unit test, in order to test the following properties:

- Formatted string is between “[ ]”
- The each element in collection, appears comma-separated
- Except the last element in collection

```
using namespace rc;
prop(
    "collection_to_str_CheckPresenceOfBrackets",
    []
    {
        using namespace utils::algorithm;
        const auto input =
*gen::arbitrary<std::vector<std::string>>();
        const auto output = collection_to_str(input);

        return (output.front() == '[') && (output.back() == ']');
    });
```

We need to include the namespace rc.
The **prop** is a wrapper around the section - that is why the first argument will describe the test.
The second argument is callable - that will be executed within the section.
The callable should return boolean - as indication on the test outcome.
Rapidcheck framework provides different generators - gen::arbitrary<T> will return the Generator<T>: a random generator of T, where the seed and the number of generated values (default 100) is matter of configuration

Ok - let’s implement the test for second property: counting the number of commas

```
const auto commas_counter = [](const std::string& str)
{
    int count = 0;

    auto begin = str.begin();
    const auto end = str.end();

    const char separator = ',';

    for (;;)
    {
        if (begin == end) { break; }
        const auto comma = std::find(begin, end, separator);
        if (comma == end) { break; }
        ++count;
        begin = std::next(comma);
```

```
    }

    return count;
};


prop(
    "collection_to_str_CheckNumberOfCommas",
    [&commas_counter]
    {
        using namespace utils::algorithm;
        const auto input = *gen::suchThat(
            gen::nonEmpty<std::vector<std::string>>(),
            [](const auto& vec) {  // filter: input string doesn't
contain any commas
                return std::all_of(
                    vec.cbegin(),
                    vec.cend(),
                    [](const auto& str) { return not str.empty() &&
str.find(',') == std::string::npos; });
            });

        const auto output = collection_to_str(input);

        return commas_counter(output) == input.size() - 1;
    });
```

Here we need to employ some test logic:

- We need to implement the helper method for counting the commas in output string
- We need to **customize the generator** - using **gen::suchThat** which takes as a first argument the generator - non-empty collection of strings (inputs), and as second argument the unary predicate: that will be used to filter the generated inputs.
  In our case - for testing scenario to make sense, we will filter the elements of the collection (strings) that contain the separator.

Finally, the last property test - evaluates the last element, not being comma-separated

```
const auto no_comma_after_last_element = [](const std::string&
formattedOutput, std::size_t lastElementSize)
{
    const auto lastComma = formattedOutput.find_last_of(',');
    // no comma found: single element collection
    if (lastComma == std::string::npos) { return true; }
    // [..., lastElement] - comma + blank space
```

```
    const auto lastElementPosition = lastComma + 2;
    return formattedOutput.at(lastElementPosition + lastElementSize)
== ']';
};

prop(
    "collection_to_str_CheckLastElementNoSeparator",
    [&no_comma_after_last_element]
    {
        using namespace utils::algorithm;
        const auto input = *gen::suchThat(
            gen::nonEmpty<std::vector<std::string>>(),
            [](const auto& vec) {  // filter: non-empty input string
doesn't contain any commas
                return std::all_of(
                    vec.cbegin(),
                    vec.cend(),
                    [](const auto& str) { return not str.empty() &&
str.find(',') == std::string::npos; });
            });

        const auto output = collection_to_str(input);
        return no_comma_after_last_element(output,
input.back().size());
    });
```

For the completeness, here are the helper methods - for implementing collection formatting (C++17), handling the special cases

```
template <typename Byte>
constexpr bool is_byte_type =
std::is_same_v<Byte, std::uint8_t> ||
std::is_same_v<Byte, std::int8_t>  ||
std::is_same_v<Byte, unsigned char> ||
std::is_same_v<Byte, std::byte>;

template <typename InputIterator, typename value_type>
[[nodiscard]] auto byte_collection_to_str(InputIterator first,
InputIterator last)
    -> std::enable_if_t<is_byte_type<value_type>, std::string>
{
    if (first == last) [[unlikely]]
        return std::string("[]");
    assert(first < last);  // wrong parameters
```

```
    std::stringstream ss;
    ss << '[';

    using conv_value_type =
std::conditional_t<std::is_signed_v<value_type>, std::int32_t,
std::uint32_t>;
    ss << std::hex << std::uppercase;  // Set hex format and uppercase
    std::for_each(first, std::prev(last), [&ss](auto& value) {
        ss << "0x" << std::setw(2) << std::setfill('0')
           << static_cast<conv_value_type>(value) << ", ";
    });
    ss << "0x" << std::setw(2) << std::setfill('0')
       << static_cast<conv_value_type>(*std::prev(last)) << ']';

    return ss.str();
}



template <class InputIterator>
[[nodiscard]] auto enum_collection_to_str(InputIterator first,
InputIterator last)
{
    if (first == last) [[unlikely]]
        return std::string("[]");
    assert(first < last);  // wrong parameters

    using value_type =
std::underlying_type_t<std::decay_t<decltype(*first)>>;

    // Special handling for "byte-like" types
    if constexpr (is_byte_type<value_type>)
    {
        return byte_collection_to_str<InputIterator,
value_type>(first,  last);
    }
    else
    {
        std::stringstream ss;
        ss << '[';
        std::for_each(first, std::prev(last),
        [&ss](auto& value)
        { ss << static_cast<value_type>(value) << ", "; });
        ss << static_cast<value_type>(*std::prev(last)) << ']';
```

```
        return ss.str();
    }
}

template <typename InputIterator>
[[nodiscard]] auto basic_collection_to_str(InputIterator first,
InputIterator last)
{
    if (first == last) [[unlikely]]
        return std::string("[]");
    assert(first < last);  // wrong parameters

    using value_type = std::decay_t<decltype(*first)>;

    std::stringstream ss;

    ss << '[';
    std::copy(first, std::prev(last),
std::ostream_iterator<value_type>(ss, ", "));
    ss << *std::prev(last) << ']';

    return ss.str();
}
```