ApprovalTests.cpp

Llewellyn Falco, Clare Macrae

Jan 23, 2024

GETTING STARTED

ApprovalTests.cpp is a C++ implementation of Approval Tests.

Also known as Golden Master Tests or Snapshot Testing, Approval Tests are an alternative to asserts. They are great for testing objects with lots of fields, or lists of objects.

For releases and source code, see GitHub.

CHAPTER

ONE

GETTING STARTED

If you are new to Approval Tests, or to this C++ library, start here:

- **Concepts**: *Overview* | *ApprovalTesting (the concept)* | *The Path to Approval Testing*
- **Start coding**: *Tutorial* | *Setup Options* | *Choosing a test framework* | *Approving Results*

1.1 Overview of Approval Tests

1.1.1 Summary

Approval Tests are an alternative to writing expression assertions in your code.

As the following examples demonstrate, Approval Tests can result in significantly simpler and more elegant tests of complex objects.

1.1.2 Traditional Asserts

Traditional tests spend equal time focusing on creating the inputs and verifying the outputs.

When the objects being tested are non-trivial, either the tests become quite verbose (as shown in this example), or it's tempting to only test a small part of the behaviour.

```
// Arrange, Act
Sandwich s = createSandwichForTest();
// Assert
REQUIRE("Sourdough" == s.getBread());
REQUIRE(s.getCondiments().contains("Mayo"));
REQUIRE(s.getCondiments().contains("Pepper"));
REQUIRE(s.getCondiments().contains("Olive Oil"));
REQUIRE(s.getFillings().contains("Tomato"));
REQUIRE(s.getFillings().contains("Lettuce"));
REQUIRE(s.getFillings().contains("Cheddar"));
```

(See snippet source)

1.1.3 Approval Tests

Approval Tests simplify the verification of outputs. They do this by writing the outputs to a file, which once saved, becomes your spec.

You still supply the inputs, but Approval Tests gives you powerful ways of viewing complex outputs, meaning you can move on to the next feature or next test more quickly.

```
// Arrange, Act
Sandwich s = createSandwichForTest();
// Assert
ApprovalTests::Approvals::verify(s);
```

(See snippet source)

This generates the approval file - which is generated **for** you, but approved by you.

```
sandwich {
    bread: "Sourdough",
    condiments: ["Mayo", "Pepper", "Olive Oil"],
    fillings: ["Tomato", "Lettuce", "Cheddar"]
}
```

(See snippet source)

1.2 ApprovalTesting (the concept)

This is also referred to as snapshot testing, or golden master testing.

1.2.1 The Idea

Most tests have you explicitly state what is expected before you write the code, ApprovalTesting, instead has you state the data you are interested in checking, has you manually check it until you decide you are satisfied and then continues to ensure it remains consistent for the future.

ApprovalTesting also harnesses the power of other tools to make the results, and the differences in results, easier to understand and act on.

1.2.2 The Path

1 Simple verification

The technique

Most of us are familiar with simple checks for numbers or strings.

```
REQUIRE( 1 == count);
REQUIRE( "Clare" == name);
```

The problem

But this can get complicated if you have a large list or object with many fields.

```
REQUIRE( "Clare" == names[0]);
REQUIRE( "Llewellyn" == names[1]);
REQUIRE( "Simon" == names[2]);
REQUIRE( "James" == names[3]);
REQUIRE( "Emily" == names[4]);
```

One solution for this is to start writing objects to string that are easier to verify.

2 ToString verification

The technique

If we print the array in the above sample, we can instead verify the whole thing with

```
REQUIRE( "[Clare,Llewellyn,Simon,James,Emily]" == toString(names));
```

The problem

This works well, until you start to get large multi-line strings.

1.3 The Path to Approval Testing

You are undoubtedly doing some form of Approval testing already. The most basic form would be writing `REQUIRE(42 == 6 * 9)`, then running it, getting the result `42 != 54` and then changing the code to say `REQUIRE(54 == 6 * 9)`.

This is the path that we commonly see, as people move in to Approval Tests:

- Verify numbers
  - Have lots of numbers, like an array
- Strings - turn the array of numbers in to a string
  - Those strings become long
- Files - Golden Master
  - Managing them - coming up with names
- Some sort of naming convention
- Can be hard to understand what's in the files
  - Start using diff tools
- Start creating custom methods for the things you are testing
  - verifyThing

1.4 Tutorial

The tutorial is written for someone with a decent understanding of C++, a passing understanding of traditional unit testing and of diff tools, and no experience with Approval Tests at all.

In this tutorial, we are going to use Windows, Catch2 and WinMerge. If you are using something else, it will make almost no difference to your experience.

By the end of this tutorial, you should be able to use Approval Tests in most basic cases.

To follow along at home, please download the Starter Project unless you already have a working ApprovalTests build / environment in which case you can just start a new test.

1.4.1 Hello Approval Tests

Let's open the Starter Project in your development environment, and open Tutorial.cpp.

Writing the Test

Let's add our first test:

```
TEST_CASE("HelloApprovals")
{
    ApprovalTests::Approvals::verify("Hello Approvals");
}
```

(See snippet source)

Approving the Test

When we run the test, WinMerge will open as such:

On the left hand side, we will see the actual received result, `Hello Approvals`. It is what we want, so we are going to approve it. To do that, click the "All Right" button (or copy and paste the text to the other side)

Afterwards, the two sides should be identical.

Now save the file and close the Diff Tool.

Now, when you re-run the tests, two things should happen:

1. The test should pass
2. The Diff Tool should **NOT** open.

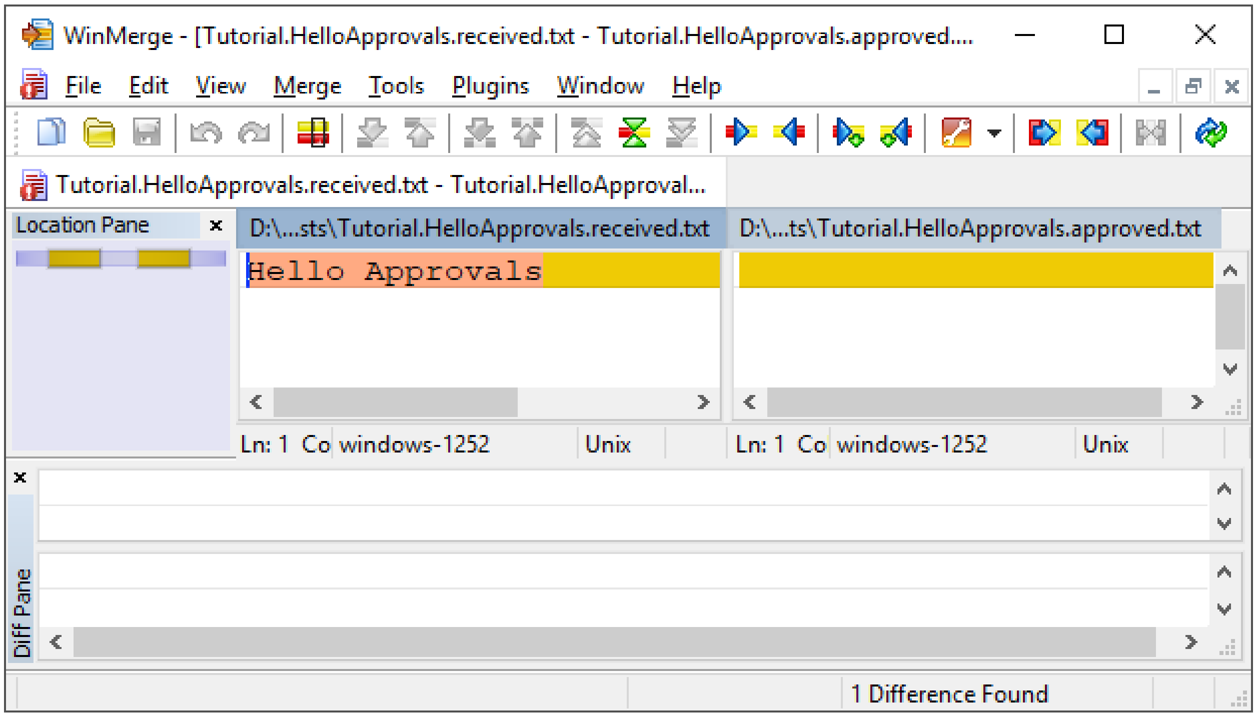
What just happened?

Approval Tests keeps its expected result in an external file. When you run the test, it reads this file to do its verification.

If it matches, the test passes, and everything is finished.

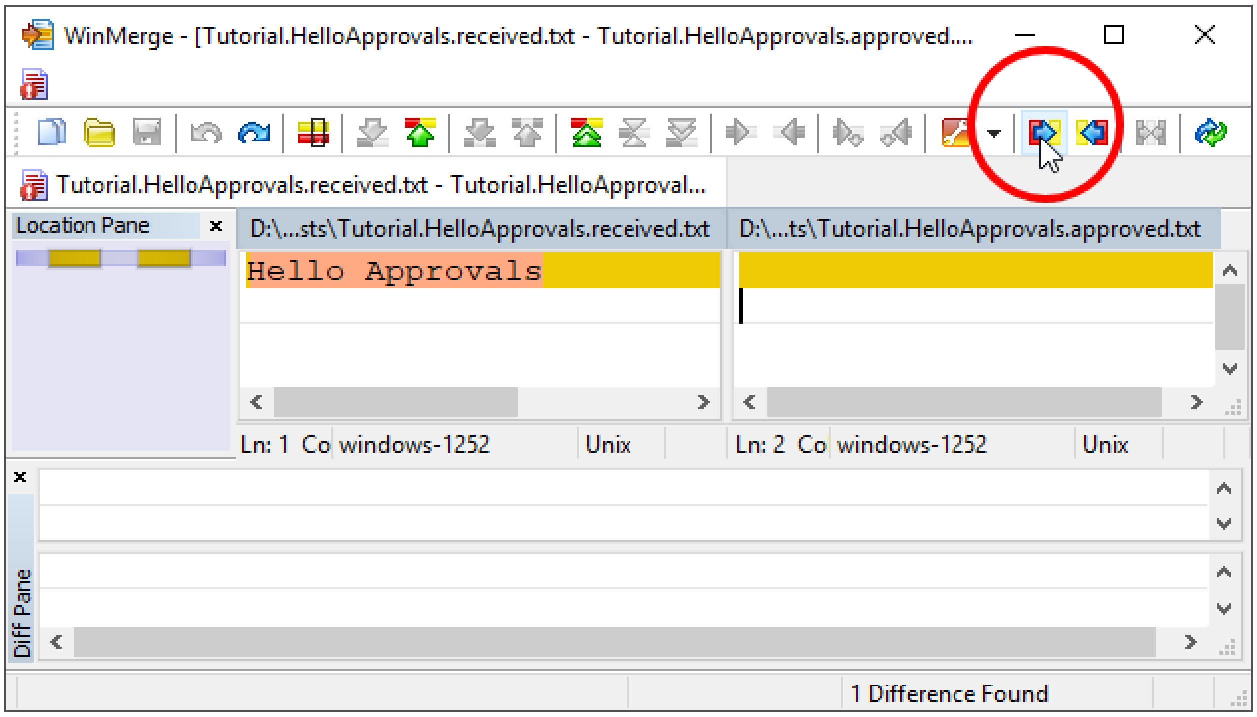
However, if it does not match, another step is invoked, and a "Reporter" (the Diff Tool) is launched. This allows you to easily view and gain insight in to what happened and decide what you want to happen.

Please note that the first time you run an Approval Test, it will always fail and launch a reporter, as you have never said anything is OK.


WinMerge - [Tutorial.HelloApprovals.received.txt - Tutorial.HelloApprovals.approved....
File Edit View Merge Tools Plugins Window Help
Tutorial.HelloApprovals.received.txt - Tutorial.HelloApproval...
Location Pane
D:\...sts\Tutorial.HelloApprovals.received.txt
D:\...ts\Tutorial.HelloApprovals.approved.txt
Ln: 1 Co windows-1252 Unix
Ln: 1 Co windows-1252 Unix
Diff Pane
1 Difference Found

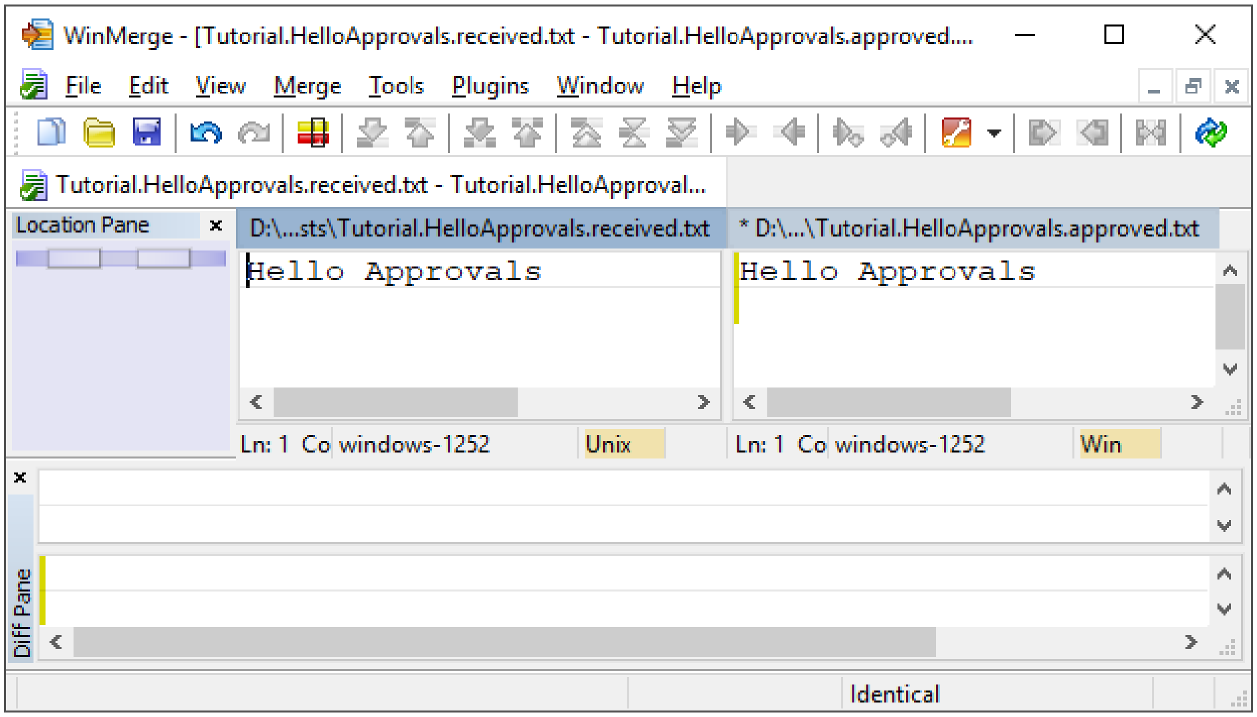
```
Hello Approvals
```

Fig. 1: New Failure


WinMerge - [Tutorial.HelloApprovals.received.txt - Tutorial.HelloApprovals.approved....
Tutorial.HelloApprovals.received.txt - Tutorial.HelloApproval...
Location Pane
D:\...sts\Tutorial.HelloApprovals.received.txt
D:\...ts\Tutorial.HelloApprovals.approved.txt
Ln: 1 Co windows-1252 Unix
Ln: 2 Co windows-1252 Unix
Diff Pane
1 Difference Found

```
Hello Approvals
```

Fig. 2: Approving


WinMerge - [Tutorial.HelloApprovals.received.txt - Tutorial.HelloApprovals.approved....
File Edit View Merge Tools Plugins Window Help
Tutorial.HelloApprovals.received.txt - Tutorial.HelloApproval...
Location Pane
D:\...sts\Tutorial.HelloApprovals.received.txt
* D:\...\Tutorial.HelloApprovals.approved.txt
Ln: 1 Co windows-1252 Unix
Ln: 1 Co windows-1252 Win
Diff Pane
Identical

```
Hello Approvals
```

```
Hello Approvals
```

Fig. 3: Approved

Approval Files

Approvals creates a lot of `.approved.txt` and `.received.txt` files. The `.received.txt` files are automatically deleted on a passing test, and should never be checked in to source control. We suggest adding `*.received.*` line to your `.gitignore` file.

The `.approved.txt` files, on the other hand, need to be checked in to your source control.

Approval Tests follows the Convention over Configuration rule. The convention used for our files is as follows:

`FileName.TestName.approved.txt`

So in this case, it will be:

`Tutorial.HelloApprovals.approved.txt`

It will be located in the same directory as your tests. (This is configurable).

1.4.2 The ApprovalTests namespace

In all other code examples in this site, have already included the code:

```cpp
using namespace ApprovalTests;
```

(See snippet source)

… So that code samples are simpler and easier to read. This is a recommended practice in your tests.

1.4.3 Approving Objects

The above example is a bit simplistic. Normally, you will want to test actual objects from your code base. To explore this, let's create an object called `LibraryBook`:

```
class LibraryBook
{
public:
    LibraryBook(std::string title_,
                std::string author_,
                int available_copies_,
                std::string language_,
                int pages_,
                std::string isbn_)
        : title(title_)
        , author(author_)
        , available_copies(available_copies_)
        , language(language_)
        , pages(pages_)
        , isbn(isbn_)
    {
    }
    // Data public for simplicity of test demo case.
    // In production code, we would have accessors instead.
    std::string title;
    std::string author;
    int available_copies;
    std::string language;
    int pages;
    std::string isbn;
};
```

(See snippet source)

What we would like to be able to write is:

```
LibraryBook harry_potter(
    "Harry Potter and the Goblet of Fire", "J.K. Rowling",
    30, "English", 752, "978-0439139595");

Approvals::verify(harry_potter); // This does not compile
```

(See snippet source)

The problem is that this will not compile, because at present there is no way to turn the LibraryBook in to a string representation.

So we are going to add a lambda to handle the printing.

Let's start by just printing the title:

```
Approvals::verify(harry_potter, [](const LibraryBook& b, std::ostream& os) {
    os << "title: " << b.title;
});
```

(See snippet source)

There's a lot going on here, so let's break it down:

1. Lambda: `[](const LibraryBook& b, std::ostream& os){}`. This is the call-back function to convert your object to a string. Note that you can also write this as `[](auto b, auto& os){}`
2. toString: `os << "title:  " << b.title;` - this is the bit of code that actually turns our object in to a string.

This works, but of course, there is a lot more that we want to look at than the title. So let's expand the `toString`:

```
Approvals::verify(harry_potter, [](const LibraryBook& b, std::ostream& os) {
    os << "title: " << b.title << "\n"
       << "author: " << b.author << "\n"
       << "available_copies: " << b.available_copies << "\n"
       << "language: " << b.language << "\n"
       << "pages: " << b.pages << "\n"
       << "isbn: " << b.isbn << "\n";
});
```

(See snippet source)

When you run and approve this, you will end up with the approval file:

```
title: Harry Potter and the Goblet of Fire
author: J.K. Rowling
available_copies: 30
language: English
pages: 752
isbn: 978-0439139595
```

(See snippet source)

If you would like to know how to do this more robustly, check out To String.

1.4.4 Dealing with test failures

Every change in behaviour is not necessarily a failure, but every change in behaviour will fail the test.

There are three parts to dealing with failure.

1. Identify what changed
2. Either:

- Fix the code, if the change was not intentional
- Re-approve the test, if you want the new behaviour

If you are in a refactoring mode, changes in Approval Tests output files are usually unintended, and a sign that you might have made a mistake.

If you are adding a new feature, changes in Approval Tests output files are often intended, and a sign that you should review and maybe accept the modified output.

1.4.5 Demo

Here's a little video of the whole process.

Fig. 4: Intro Graphic

1.5 Setup Options

There are three different places you might be starting your setup from.

Here's how to set up from:

1.5.1 I have nothing

This is when you are interested in how Approval Tests work and want to see Approval Tests in action using our provided examples.

If you just want to start doing some TDD with Approval Tests, the Tutorial will walk you through the following broad steps:

1. download the ApprovalTests.cpp.StarterProject
2. download a Supported Diff Tool
3. Run the existing tests from the project (they should pass).
4. Then open the file tests/NewTest.cpp, choose which of the three starting points you want, and Go!

1.5.2 I have code but no tests

This is when you have an existing code base that you would like to start testing, as you don't yet have any tests at all.

Start by choose a testing framework.

Then continue to the next section…

1.5.3 I have tests and code and want to add Approval Tests

This is when you have an existing code base that has some tests already, and you would like to improve test coverage.

1. Download the single header file
2. Do one of:
   - Add a wrapper header
   - Rename the header file to `ApprovalTests.hpp` (removing its version number)
3. Set up your `main()` (this is framework-dependent):
   - Using Approval Tests With Catch
   - Using Approval Tests With CppUTest
   - Using Approval Tests With Google Tests
   - Using Approval Tests With Doctest

   - Using Approval Tests With Boost.Test
   - Using Approval Tests With [Boost].UT
4. Download a Supported Diff Tool

If you current test framework is not supported, see Supporting a new test framework

Next Steps

- Try the Tutorial

1.6 Getting Started - Creating your main()

1.6.1 Introduction

This page shows how to set up the `main()` for test programs that use Approval Tests.

These steps are needed in order to teach Approval Tests how to name its output files automatically.

If, after following these steps, you need help with running your program, please see Troubleshooting.

1.6.2 Main File

The Basics

You need to include 2 lines for your main file to work.

For Catch2, it's these two lines:

```
// main.cpp:
#define APPROVALS_CATCH // This tells Approval Tests to provide a main() - only do this␣
↪in one cpp file
#include "ApprovalTests.hpp"
```

(See snippet source)

For all other test files, you need:

```
#include "ApprovalTests.hpp"
```

Details

- Using Approval Tests With Catch
- Using Approval Tests With CppUTest
- Using Approval Tests With Google Tests
- Using Approval Tests With Doctest
- Using Approval Tests With Boost.Test
- Using Approval Tests With [Boost].UT

1.6.3 Choosing a testing framework

If you are already using one of the above testing frameworks, that is the one you should use.

If not, Approval Tests works well with all the above. Here are factors to consider.

Framework	Min C++	Ease of setup	IDE Support	Build time
Catch2	C++11	Very easy [1], [2]	Widely supported	Not bad [3]
CppUTest	C++11	Difficult	Unknown	Very Fast
doctest	C++11	Very easy [1]	Unknown	Fast
Google Test	C++11	Difficult	Very widely supported	Fast
Boost.Test	C++11	Difficult	Unknown	Fast
[Boost].UT	C++20 [4]	Very easy [1]	Unknown	Fast

1. Released as a single header file
2. See the Starter Project
3. Catch2 has options to speed up its builds
4. [Boost].UT works with C++17, but the ApprovalTests interface to that library depends on std::source_location, which is a C++ 20 feature.

1.7 Approving Results

1.7.1 Concept

Approving results just means saving the .approved file with your desired results.

We'll provide more explanation in due course, but, briefly, here are the most common approaches to do this.

Via Diff Tool

Most diff tools have the ability to move text from left to right, and save the result.

Via command line

If you use the clipboard reporter, it will create a command line to move the received file to the approved file for you.

Limitation: only one command at a time, currently.

Via file rename

You can just rename the .received file to .approved.

CHAPTER

TWO

TEST FRAMEWORKS

Approval Tests uses a test framework, in order to find out the names of tests and of source files. The test framework will also report errors for any failed Approval Tests.

- **Using Approval Tests with**: *Boost.Test* | *Catch2* | *CppUTest* | *doctest* | *Google Tests* | *[Boost].ut*
- **Extending test framework support**: *Supporting a new test framework*

2.1 Using Approval Tests With Boost.Test

2.1.1 Introduction

The Boost.Test test framework works well with Approval Tests.

Note: this document assumes the reader is familiar with the Boost.Test framework.

2.1.2 Requirements

Approval Tests for Boost.Test requires that you specify the `#include <.../unit_test.hpp>`
This allows ApprovalTests to work with all the different configurations of boost.

Approval Tests needs Boost.Test version 1.60.0 or above.

2.1.3 Getting Started With Boost.Test

Adding ApprovalTests to your Boost.Test

To enable any Boost.Test test files to use ApprovalTests, find the corresponding entry point and
add the following lines of code to your Test module's entry point after the boost headers:

```
// test_entry_point.cpp file[s] (after #including boost.test)
#define APPROVALS_BOOSTTEST
#include "ApprovalTests.hpp"
```

(See snippet source)

Understanding Boost.Test Entry points

A directory of Boost.Test source files can either have multiple or a single entry point[s]. The entry point is any file that will contain the line:

```
#define BOOST_TEST_MODULE ModuleName
```

(See snippet source)

2.1.4 Code to copy for your first Boost.Test Approvals test

Here is sample code to create your `main()` function, to set up Approval Tests' Boost.Test integration.

We called this file `boost_starter_main.cpp`:

```
#define BOOST_TEST_MODULE ModuleName

//#include <boost/test/unit_test.hpp> // static or dynamic boost build
#include <boost/test/included/unit_test.hpp> // header only boost

#define APPROVALS_BOOSTTEST
#include "ApprovalTests.hpp"

// This puts "received" and "approved" files in approval_tests/ sub-directory,
// keeping the test source directory tidy:
auto directoryDisposer =
    ApprovalTests::Approvals::useApprovalsSubdirectory("approval_tests");
```

(See snippet source)

Here is sample code to create your first test. We called this file `boost_starter_test.cpp`:

```
#define BOOST_TEST_INCLUDED
#include <boost/test/unit_test.hpp>

#include "ApprovalTests.hpp"

BOOST_AUTO_TEST_SUITE(SuiteName)
BOOST_AUTO_TEST_CASE(TestCaseName)
{
    ApprovalTests::Approvals::verify(42);
}
BOOST_AUTO_TEST_SUITE_END()
```

(See snippet source)

And finally, here is sample code to put in your `CMakeLists.txt` file:

```
set(EXE_NAME boost_starter)
set(CMAKE_CXX_STANDARD 11)

find_package(Boost 1.60.0 COMPONENTS REQUIRED)
if (NOT Boost_FOUND)
    message(FATAL_ERROR "Cannot find Boost libraries")
```

(continues on next page)

(continued from previous page)

```
endif ()

include_directories(SYSTEM ${Boost_INCLUDE_DIRS})

add_executable(${EXE_NAME}
        boost_starter_main.cpp
        boost_starter_test.cpp
        )
target_link_libraries(${EXE_NAME} ApprovalTests::ApprovalTests ${Boost_LIBRARIES})

add_test(NAME ${EXE_NAME} COMMAND ${EXE_NAME})
```

(See snippet source)

2.2 Using Approval Tests With Catch

2.2.1 Introduction

The Catch2 test framework works well with Approval Tests.

This section describes the various ways of using Approval Tests with Catch2.

Notes pre-v.10.8.0:

Earlier versions of Approval Tests had issues with Ninja. Read more at Troubleshooting Misconfigured Build.

2.2.2 Requirements

Approval Tests requires that a file called the following is found:

```
#include <catch2/catch.hpp>
```

(See snippet source)

(Before v7.0.0, it required `Catch.hpp`)

2.2.3 Getting Started With Catch2

Starter Project

The quickest way to start experimenting with Approval Tests is to:

1. Download the project ApprovalTests.cpp.StarterProject - via the green “Clone or Download” button at the top-right of the project site.
2. Opening the project in the C++ IDE of your choice.

Each time we release a new version of Approval Tests, we update this project, so it always has the latest features.

New Project

Create a file `main.cpp` and add just the following two lines:

```
// main.cpp:
#define APPROVALS_CATCH // This tells Approval Tests to provide a main() - only do this␣
↪in one cpp file
#include "ApprovalTests.hpp"
```

(See snippet source)

Existing Project - with CATCH_CONFIG_MAIN

If you have a Catch2 project with your own `main.cpp` that contains the following lines, you will need to replace them with the code in the previous section.

```
#define CATCH_CONFIG_MAIN // remove these lines, and replace with Approval Tests lines
#include "catch2/catch.hpp"
```

Existing Project - with your main()

If you have supplied your own main() for Catch, you will need to teach it how to supply test names to Approval Tests.

You should make the following additions to your own source file that contains `main()`.

```
// Add these two lines to the top of your main.cpp file:
#define APPROVALS_CATCH_EXISTING_MAIN
#include "ApprovalTests.hpp"
```

(See snippet source)

2.2.4 Code to copy for your first Catch2 Approvals test

Here is sample code to create your `main()` function, to set up Approval Tests' Catch2 integration.

We called this file `catch2_starter_main.cpp`:

```
#define APPROVALS_CATCH
#include "ApprovalTests.hpp"

// This puts "received" and "approved" files in approval_tests/ sub-directory,
// keeping the test source directory tidy:
auto directoryDisposer =
    ApprovalTests::Approvals::useApprovalsSubdirectory("approval_tests");
```

(See snippet source)

Here is sample code to create your first test. We called this file `catch2_starter_test.cpp`:

```
#include "catch2/catch.hpp"
#include "ApprovalTests.hpp"

TEST_CASE("catch2_starter sample")
```

(continues on next page)

(continued from previous page)

```
{
    // TODO Replace 42 with the value or object whose contents you are verifying.
    // For help, see:
    // https://approvaltestscpp.readthedocs.io/en/latest/generated_docs/ToString.html
    ApprovalTests::Approvals::verify(42);
}
```

(See snippet source)

And finally, here is sample code to put in your `CMakeLists.txt` file:

```
set(EXE_NAME catch2_starter)
set(CMAKE_CXX_STANDARD 11)
add_executable(${EXE_NAME}
        catch2_starter_main.cpp
        catch2_starter_test.cpp
        )
target_link_libraries(${EXE_NAME} ApprovalTests::ApprovalTests Catch2::Catch2)

add_test(NAME ${EXE_NAME} COMMAND ${EXE_NAME})
```

(See snippet source)

2.3 Using Approval Tests With CppUTest

2.3.1 Introduction

The CppUTest test framework works well on most platforms with Approval Tests.

This section describes the various ways of using Approval Tests with CppUTest.

Notes pre-v.10.8.0:

Earlier versions of Approval Tests had issues with Ninja. Read more at Troubleshooting Misconfigured Build.

CppUTest Integration Limitations

Note: This integration is not tested on CygWin. The CppUTest integration with Approval Tests does not build on this platform, CygWin, therefore our tests of it are disabled.

Note: Approval Tests's use of STL objects triggers test failures from CppUTest's memory-leak checking, and so our integration with CppUTest currently **turns of its memory leak checks**.

2.3.2 Requirements

Approval Tests requires that the following are found:

```
#include <CppUTest/CommandLineTestRunner.h>
#include <CppUTest/TestPlugin.h>
#include <CppUTest/TestRegistry.h>
```

(See snippet source)

Approval Tests is tested with CppUTest v4.0.

2.3.3 Getting Started With CppUTest

Starter Project

We haven't yet provided a Starter Project for using Approval Tests with CppUTest.

This is partly based on the assumption that anyone already using CppUTest will have their own project set up, and anyone else would probably use Catch2 instead.

If it would be helpful for us to such a Starter Project, please let us know, via the contact details in Contributing to ApprovalTests.cpp.

New Project

Create a file `main.cpp` and add just the following two lines:

```
#define APPROVALS_CPPUTEST
#include "ApprovalTests.hpp"
```

(See snippet source)

Existing Project - with your main()

If you have an existing CppUTest-based test program that provides its own `main()`, you won't be able to use the approach above.

Instead, you should make the following additions to your own source file that contains `main()`.

```
// main.cpp:

// 1. Add these two lines to your main:
#define APPROVALS_CPPUTEST_EXISTING_MAIN
#include "ApprovalTests.hpp"

int main(int argc, char** argv)
{
    // 2. Add this line to your main:
    ApprovalTests::initializeApprovalTestsForCppUTest();

    int result = CommandLineTestRunner::RunAllTests(argc, argv);
    TestRegistry::getCurrentRegistry()->resetPlugins();
```

(continues on next page)

(continued from previous page)

```
    return result;
}
```

(See snippet source)

2.3.4 Code to copy for your first CppUTest Approvals test

Here is sample code to create your `main()` function, to set up Approval Tests' CppUTest integration.

We called this file `cpputest_starter_main.cpp`:

```
#define APPROVALS_CPPUTEST
#include "ApprovalTests.hpp"

// This puts "received" and "approved" files in approval_tests/ sub-directory,
// keeping the test source directory tidy:
auto directoryDisposer =
    ApprovalTests::Approvals::useApprovalsSubdirectory("approval_tests");
```

(See snippet source)

Here is sample code to create your first test. We called this file `cpputest_starter_test.cpp`:

```
#include "ApprovalTests.hpp"
#include "CppUTest/TestHarness.h"

TEST_GROUP(CppUTestStarter){};

TEST(CppUTestStarter, Sample)
{
    // TODO Replace 42 with the value or object whose contents you are verifying.
    // For help, see:
    // https://approvaltestscpp.readthedocs.io/en/latest/generated_docs/ToString.html
    ApprovalTests::Approvals::verify(42);
}
```

(See snippet source)

And finally, here is sample code to put in your `CMakeLists.txt` file:

```
set(EXE_NAME cpputest_starter)
set(CMAKE_CXX_STANDARD 11)
add_executable(${EXE_NAME}
        cpputest_starter_main.cpp
        cpputest_starter_test.cpp
        )
target_link_libraries(${EXE_NAME} ApprovalTests::ApprovalTests CppUTest)

add_test(NAME ${EXE_NAME} COMMAND ${EXE_NAME})
```

(See snippet source)

2.4 Using Approval Tests With doctest

2.4.1 Introduction

The doctest test framework works well with Approval Tests.

Notes pre-v.10.8.0:

Earlier versions of Approval Tests had issues with Ninja. Read more at Troubleshooting Misconfigured Build.

Doctest is similar to Catch, but claims to give faster compilation times.

2.4.2 Requirements

Approval Tests for doctest requires that a file called the following is found:

```
#include <doctest/doctest.h>
```

(See snippet source)

Approval Tests needs doctest version 2.3.4 or above.

2.4.3 Getting Started With doctest

New Project

Create a file `main.cpp` and add just the following two lines:

```
// main.cpp:
#define APPROVALS_DOCTEST // This tells Approval Tests to provide a main() - only do␣
↪this in one cpp file
#include "ApprovalTests.hpp"
```

(See snippet source)

Existing Project - with your main()

If you have supplied your own main() for doctest, you will need to teach it how to supply test names to Approval Tests.

You should make the following additions to your own source file that contains `main()`.

```
// Add these two lines to the top of your main.cpp file:
#define APPROVALS_DOCTEST_EXISTING_MAIN
#include "ApprovalTests.hpp"
```

(See snippet source)

2.4.4 Code to copy for your first doctest Approvals test

Here is sample code to create your `main()` function, to set up Approval Tests' doctest integration.

We called this file `doctest_starter_main.cpp`:

```cpp
#define APPROVALS_DOCTEST
#include "ApprovalTests.hpp"

// This puts "received" and "approved" files in approval_tests/ sub-directory,
// keeping the test source directory tidy:
auto directoryDisposer =
    ApprovalTests::Approvals::useApprovalsSubdirectory("approval_tests");
```

(See snippet source)

Here is sample code to create your first test. We called this file `doctest_starter_test.cpp`:

```cpp
#include "doctest/doctest.h"
#include "ApprovalTests.hpp"

TEST_CASE("doctest_starter sample")
{
    // TODO Replace 42 with the value or object whose contents you are verifying.
    // For help, see:
    // https://approvaltestscpp.readthedocs.io/en/latest/generated_docs/ToString.html
    ApprovalTests::Approvals::verify(42);
}
```

(See snippet source)

And finally, here is sample code to put in your `CMakeLists.txt` file:

```cmake
set(EXE_NAME doctest_starter)
set(CMAKE_CXX_STANDARD 11)
add_executable(${EXE_NAME}
        doctest_starter_main.cpp
        doctest_starter_test.cpp
        )
target_link_libraries(${EXE_NAME} ApprovalTests::ApprovalTests doctest::doctest)

add_test(NAME ${EXE_NAME} COMMAND ${EXE_NAME})
```

(See snippet source)

2.5 Using Approval Tests With Google Tests

2.5.1 Introduction

The Google Test test framework works well with Approval Tests.

This section describes the various ways of using Approval Tests with Google Test.

Notes pre-v.10.8.0:

Earlier versions of Approval Tests had issues with Ninja. Read more at Troubleshooting Misconfigured Build.

2.5.2 Getting Started With Google Test

Starter Project

We haven't yet provided a Starter Project for using Approval Tests with Google Tests.

This is partly based on the assumption that anyone already using Google Tests will have their own project set up, and anyone else would probably use Catch2 instead.

If it would be helpful for us to such a Starter Project, please let us know, via the contact details in Contributing to ApprovalTests.cpp.

New Project

Create a file `main.cpp` and add just the following two lines:

```
// main.cpp:
#define APPROVALS_GOOGLETEST // This tells Approval Tests to provide a main() - only do␣
↪this in one cpp file
#include "ApprovalTests.hpp"
```

(See snippet source)

Existing Project - no main()

Google Test has a `gtest_main` library that provides a `main()` function, and then runs all your tests.

If your existing Google Test application uses the `gtest_main` library, Approval Tests will not be able to obtain the names to use output files. You will then see the help message shown in Troubleshooting.

To fix this, please add a new `main.cpp`, as shown in the previous section.

Existing Project - with your main()

If you have an existing Google Test-based test program that provides its own `main()`, you won't be able to use the approach above.

Instead, you should make the following additions to your own source file that contains `main()`.

```
// main.cpp:

// 1. Add these two lines to your main:
#define APPROVALS_GOOGLETEST_EXISTING_MAIN
#include "ApprovalTests.hpp"

int main(int argc, char** argv)
{
    ::testing::InitGoogleTest(&argc, argv);

    // 2. Add this line to your main:
    ApprovalTests::initializeApprovalTestsForGoogleTests();

    return RUN_ALL_TESTS();
}
```

(See snippet source)

2.5.3 Customizing Google Tests Approval File Names

Most testing frameworks have two pieces of naming information: `SourceFileName` and `TestName`.

Google Tests has an additional piece of information: `TestCaseName`.

```
TEST(TestCaseName, TestName)
```

(See snippet source)

With Google Tests, this will result in Approvals creating output files beginning with:

```
SourceFileName.TestCaseName.TestName
```

Very often, the `SourceFileName` and the `TestCaseName` are redundant, meaning that what you would like is:

```
SourceFileName.TestName
```

By default, Approval Tests will do this if `TestCaseName` is completely contained within `SourceFileName`, meaning it is a sub-string.

Customizing

If this is not enough, Approvals allows for customization, in two ways.

Note: to be able to add these pieces of code outside of a function, you need to hold on to the result as a variable. This variable is not used, it is only there to allow the method to execute.

Note: using these customizations inside a Google `TEST` or `TEST_F`, is too late for that test: they won't take effect until the next executed test.

Note: this customization is permanent: it affects all tests run later in the current program run.

Note: this customization is cannot be reversed.

Custom Suffixes

For example, if you are Google test fixtures, you might have a lot of class names of the format `TestClassNameFixture`. You can tell Approval Tests that these are the same by adding the following to your main:

```
// main.cpp
auto customization =
    ApprovalTests::GoogleConfiguration::addIgnorableTestCaseNameSuffix("Fixture");
```

(See snippet source)

Custom Anything

If you have something more unique, you can write a function that will match if the test case name and the source file names should be considered equal.

For example, let's say you want a special tag `IgnoreThis` to indicate a that a TestCaseName should be ignored, when determining the names of output files.

So:

```
TEST(TestCaseName_IgnoreThis, TestName)
```

(See snippet source)

Would produce an output file beginning with:

```
auto outputFileBaseName = "GoogleFixtureNamerCustomizationTests.TestName";
```

(See snippet source)

You could achieve this by registering a function pointer like this:

```
// main.cpp
bool dropTestCaseNamesWithIgnoreThis(const std::string& /*testFileNameWithExtension*/,
                                     const std::string& testCaseName)
{
    return ApprovalTests::StringUtils::contains(testCaseName, "IgnoreThis");
}

auto ignoreNames = ApprovalTests::GoogleConfiguration::addTestCaseNameRedundancyCheck(
    dropTestCaseNamesWithIgnoreThis);
```

(See snippet source)

Or by using a lambda like this:

```
// main.cpp
auto ignoreNamesLambda =
    ApprovalTests::GoogleConfiguration::addTestCaseNameRedundancyCheck(
        [](const std::string& /*testFileNameWithExtension*/,
           const std::string& testCaseName) {
            return ApprovalTests::StringUtils::contains(testCaseName, "IgnoreThis");
        });
```

(See snippet source)

2.5.4 Code to copy for your first Google Test Approvals test

Here is sample code to create your `main()` function, to set up Approval Tests' Google Test integration.

We called this file `googletest_starter_main.cpp`:

```
#define APPROVALS_GOOGLETEST
#include "ApprovalTests.hpp"

// This puts "received" and "approved" files in approval_tests/ sub-directory,
```

(continues on next page)

(continued from previous page)

```
// keeping the test source directory tidy:
auto directoryDisposer =
    ApprovalTests::Approvals::useApprovalsSubdirectory("approval_tests");
```

(See snippet source)

Here is sample code to create your first test. We called this file `googletest_starter_test.cpp`:

```
#include "gtest/gtest.h"
#include "ApprovalTests.hpp"

TEST(GoogleTestStarter, Sample)
{
    // TODO Replace 42 with the value or object whose contents you are verifying.
    // For help, see:
    // https://approvaltestscpp.readthedocs.io/en/latest/generated_docs/ToString.html
    ApprovalTests::Approvals::verify(42);
}
```

(See snippet source)

And finally, here is sample code to put in your `CMakeLists.txt` file:

```
set(EXE_NAME googletest_starter)
set(CMAKE_CXX_STANDARD 11)
add_executable(${EXE_NAME}
        googletest_starter_main.cpp
        googletest_starter_test.cpp
        )
target_link_libraries(${EXE_NAME} ApprovalTests::ApprovalTests gtest gtest_main)

add_test(NAME ${EXE_NAME} COMMAND ${EXE_NAME})
```

(See snippet source)

2.6 Using Approval Tests With [Boost].UT

2.6.1 Introduction

The [Boost].UT test framework works well with Approval Tests.

[Boost].UT is a single header/single module, macro-free (micro)/Unit Testing Framework that requires C++17 / C++20

Notes pre-v.10.8.0:

Earlier versions of Approval Tests had issues with Ninja. Read more at Troubleshooting Misconfigured Build.

2.6.2 Requirements

Approval Tests for [Boost].UT requires that a file called the following is found:

```
#include <boost/ut.hpp>
```

(See snippet source)

It also requires:

- A C++ compiler that supports the C++ 20 std::source_location. See C++ compiler support.
- A build that enables C++20 - for example, with: `set(CMAKE_CXX_STANDARD 20)`
- A [Boost].UT version that is compatible with the version of ApprovalTests.cpp being used. See the version we test against: third_party/ut/include/boost/ut.hpp

2.6.3 Usage examples

Add the following two lines to your source code:

```
#define APPROVALS_UT
#include "ApprovalTests.hpp"
```

(See snippet source)

Below is an example of a call to an approval test inside a [Boost].UT test:

```
"ItCanVerifyAFile"_test = []() {
    ApprovalTests::Approvals::verify(
        "Approval Tests can verify text via the golden master method");
};
```

(See snippet source)

In the following example, two instances of ApprovalTests are called inside the same test. We need to use sections with different names, to prevent Approval Tests from using the same output file for both tests:

```
"ItCanUseMultipleVerify"_test = []() {
    {
        // Here we simulate test sections, so that Approval Tests uses different
        // output file names for the different verify() calls.
        auto section =
            ApprovalTests::NamerFactory::appendToOutputFilename("section 1");
        ApprovalTests::Approvals::verify(
            "Approval Tests can verify text via the golden master method");
    }
    {
        auto section =
            ApprovalTests::NamerFactory::appendToOutputFilename("section 2");
        ApprovalTests::Approvals::verify(
            "Approval Tests can verify different text via "
            "the golden master method");
    }
};
```

(See snippet source)

2.6.4 Code to copy for your first [Boost].UT Approvals test

Here is sample code to create your `main()` function and your first test, to set up Approval Tests' [Boost].UT integration. We called this file `ut_starter_test.cpp`:

```
#define APPROVALS_UT
#include "ApprovalTests.hpp"

int main()
{
    using namespace boost::ut;
    using namespace ApprovalTests;

    // This puts "received" and "approved" files in approval_tests/ sub-directory,
    // keeping the test source directory tidy:
    auto directory = Approvals::useApprovalsSubdirectory("approval_tests");

    "Starter"_test = []() {
        // TODO Replace 42 with the value or object whose contents you are verifying.
        // For help, see:
        // https://approvaltestscpp.readthedocs.io/en/latest/generated_docs/ToString.html
        Approvals::verify(42);
    };
}
```

(See snippet source)

And finally, here is sample code to put in your `CMakeLists.txt` file:

```
set(EXE_NAME ut_starter)

set(CMAKE_CXX_STANDARD 20)
add_executable(${EXE_NAME}
        ut_starter_test.cpp
        )
target_link_libraries(${EXE_NAME} ApprovalTests::ApprovalTests boost.ut)

add_test(NAME ${EXE_NAME} COMMAND ${EXE_NAME})
```

(See snippet source)

2.7 Supporting a new test framework

2.7.1 Introduction

ApprovalTests.cpp is designed to work with multiple C++ test frameworks.

If your test framework is not already supported, this section offers help to add that support.

2.7.2 Test Framework Requirements

ApprovalTests.cpp can be made to work with any test framework that supplies the following:

- The current test's name
- The current test's full source file path (with correct case of filename)
- Ability to report unexpected exceptions as test failures, including reporting the text in `exception.what()`, and ideally also the exception type.
- Ability to return a non-zero exit status from the test program if there were any unexpected exceptions.

2.7.3 Steps to add support

- Provide some code to add to the test's `main()` function, to listen out for the running of test cases
- Give that code a `TestName` instance, that will store information about the test being executed
- As each test case starts, update the `TestName` instance with details of the source file name, and test case name
- Add a call to `ApprovalTests::FrameworkIntegrations::setTestPassedNotification()` so that the test framework is aware that a check has been executed
- Ideally, provide a mechanism (such as a macro) that makes it easy for users to use this code in their own tests

2.7.4 Examples

This is perhaps best understood by reviewing the implementations for frameworks that are already supported - see /ApprovalTests/integrations/.

2.7.5 Adding new framework to documentation

1. Add a new file about the customisation, such as UsingCatch.md
2. Record the new framework support in:
    - The `getMisconfiguredMainHelp()` help message in HelpMessages.cpp
    - README.md - see the links to supported test frameworks in the "Requirements" section
    - include_using_test_frameworks_list.include.md
    - GettingStarted.md - see "Choosing a testing framework"
    - Setup.md - see the bullet list starting "Set up your `main()`"
    - Other documentation links: see Definition of Done

2.7.6 Update Package Managers

- Optionally, if the new library is supported by Conan, then update the ApprovalTests.cpp conan-center-index to add support for the new integration.

CHAPTER

THREE

WRITING TESTS

Now that you are set up to run Approval Tests, this section describes how to test various types of complex objects, and how to do so effectively.

- **How to Test**: *Single Objects* | *Containers* | *Combinations of containers* | *Testing Exceptions*
- **Good Practice**: *String conversions* | *Tips for Designing Strings*

3.1 Testing Single Objects

The Tutorial has a work-through of testing a single object.

3.2 Testing Containers

There are two scenarios:

- How to Test the Contents of a Container.
- How to Test a Variety of Values for One Input.

3.3 Testing Combinations

3.3.1 When to use Combinations

You have a function that takes, for example, 3 parameters, and you want to test its behaviour with a bunch of different values for each of those parameters.

If you have only one parameter that you want to vary, check out How to Test a Variety of Values for One Input.

3.3.2 Steps

1. Copy this starter text, and adjust for the number of inputs that you have.

```
TEST_CASE("CombinationsStartingPoint")
{
    std::vector<std::string> inputs1{"input1.value1", "input1.value2"};
    std::vector<std::string> inputs2{"input2.value1", "input2.value2", "input2.value3"};
    ApprovalTests::CombinationApprovals::verifyAllCombinations(
        "TITLE",
        [&](auto /*input1*/, auto /*input2*/) { return "placeholder"; },
        inputs1,
        inputs2);
}
```

(See snippet source)

2. Modify each input container for your chosen values.
3. Make sure each input type can be converted to a string (See To String)
4. Run it, and make sure that you have your inputs wired up correctly.

If they are wired up correctly, you will see a file that looks like this: it is the left hand side of the file that matters at this point: all combinations of your own input values should be listed:

```
TITLE


(input1.value1, input2.value1) => placeholder
(input1.value1, input2.value2) => placeholder
(input1.value1, input2.value3) => placeholder
(input1.value2, input2.value1) => placeholder
(input1.value2, input2.value2) => placeholder
(input1.value2, input2.value3) => placeholder
```

(See snippet source)

5. Implement the body of your lambda
6. Make sure that your lambda's return value also has an ostream operator<<
7. Change the TITLE to something meaningful
8. Run it, and approve the output.

3.3.3 The Basics

You can use `CombinationApprovals::verifyAllCombinations` to test the content of multiple containers.

This makes a kind of approval test matrix, automatically testing all combinations of a set of inputs. It's a powerful way to quickly get very good test coverage.

In this small example, all combinations of `{"hello", "world"}` and `{1, 2, 3}` are being used:

```
TEST_CASE("YouCanVerifyCombinationsOf2")
{
    std::vector<std::string> v{"hello", "world"};
```

(continues on next page)

(continued from previous page)

```
    std::vector<int> numbers{1, 2, 3};
    ApprovalTests::CombinationApprovals::verifyAllCombinations(
        [](std::string s, int i) {
            return std::string("(") + s + ", " + std::to_string(i) + ")";
        },
        v,
        numbers);
}
```

(See snippet source)

The format is carefully chosen to show both inputs and outputs, to make the test results easy to interpret. The output looks like this:

```
(hello, 1) => (hello, 1)
(hello, 2) => (hello, 2)
(hello, 3) => (hello, 3)
(world, 1) => (world, 1)
(world, 2) => (world, 2)
(world, 3) => (world, 3)
```

(See snippet source)

For advice on effective formatting, see Tips for Designing Strings. As you write out larger volumes of data in your approval files, experience has shown that the choice of layout of text in approval files can make a big difference to maintainability of tests, when failures occur.

Passing in a Reporter

Note: Over releases, the position of the optional Reporter parameter to `verifyAllCombinations` has changed, as the code has evolved:

Release	Position of optional Reporter argument
Before v.6.0.0	The optional Reporter argument goes after all the inputs
In v.6.0.0	The optional Reporter argument should be the **second** argument.
After v.6.0.0	The optional Reporter argument should be the **first** argument.
After v.8.7.0	Reporter is now stored in Options, which should be the **first** argument.

See:

- v.6.0.0 release notes
- v.7.0.0 release notes

3.3.4 C++ Language Versions

If you are using C++11, you will need to supply the exact parameter types to your lambda:

```
ApprovalTests::CombinationApprovals::verifyAllCombinations(
    [](const std::string& input1, const int input2, const double input3) {
        return functionThatReturnsSomethingOutputStreamable(input1, input2, input3);
    }, // This is the converter function
    listOfInput1s,
    listOfInput2s,
    listOfInput3s);
```

(See snippet source)

If you are using C++14 or above, you can simplify this by using `auto` or `auto&` for the lambda parameters:

```
ApprovalTests::CombinationApprovals::verifyAllCombinations(
    [](auto& input1, auto& input2, auto& input3) {
        return functionThatReturnsSomethingOutputStreamable(input1, input2, input3);
    }, // This is the converter function
    listOfInput1s,
    listOfInput2s,
    listOfInput3s);
```

(See snippet source)

3.4 Testing Exceptions

3.4.1 Testing exception messages

Testing exceptions with Approval Tests is very easy. Simply pass in a call to the function (usually wrapped in a lambda). Approval tests will execute the code, catch the exception, and verify the exception's message, i.e. `exception.what()`.

The exception thrown must inherit `std::exception`.

```
ApprovalTests::Approvals::verifyExceptionMessage([]() { /* your code goes here */ });
```

(See snippet source)

3.4.2 Handling multiple exceptions in one test

See ExceptionCollector.

3.5 String conversions

3.5.1 Introduction

When you use Approval tests, any object you pass in is going to be converted to a string. This is how Approval Tests does that, and how you can customize that behavior.

3.5.2 How Approval Tests converts your objects to strings

The process from your input to the final output looks like this - You can customize the string at any of these 4 points:

1. Input
2. The TApprovals class has a template parameter StringConverter
3. Approvals uses the default StringMaker
4. StringMaker converts via std::ostream operator (<<)

3.5.3 Pass in a string

Approval Tests can take a string, so it can be simple to simply create that string before you call `verify()`. This has the advantage of being straight-forward, but won't interact well with calls like `verifyAll()` or Combination Approvals.

3.5.4 Write a custom std::ostream operator (<<)

This is often done by providing an output operator (<<) for types you wish to test.

For example:

```
friend std::ostream& operator<<(std::ostream& os, const Rectangle1& rectangle)
{
    os << "[x: " << rectangle.x << " y: " << rectangle.y
       << " width: " << rectangle.width << " height: " << rectangle.height << "]";
    return os;
}
```

(See snippet source)

You should put this function in the same namespace as your type, or the global namespace, and have it declared before including Approval's header. (This is particularly important if you are compiling with Clang.)

If including `<iostream>` or similar is problematic, for example because your code needs to be compiled for embedded platforms, and you are tempted to surround it with `#ifdef`s so that it only shows up in testing, we recommend that you use the template approach instead:

```
template <class STREAM>
friend STREAM& operator<<(STREAM& os, const Rectangle2& rectangle)
{
    os << "[x: " << rectangle.x << " y: " << rectangle.y
       << " width: " << rectangle.width << " height: " << rectangle.height << "]";
    return os;
}
```

(See snippet source)

Wrapper classes or functions can be used to provide additional output formats for types of data:

```
struct FormatRectangleForMultipleLines
{

    explicit FormatRectangleForMultipleLines(const Rectangle3& rectangle_)
        : rectangle(rectangle_)
    {
    }

    const Rectangle3& rectangle;

    friend std::ostream& operator<<(std::ostream& os,
                                    const FormatRectangleForMultipleLines& wrapper)
    {
        os << "(x,y,width,height) = (" << wrapper.rectangle.x << ","
           << wrapper.rectangle.y << "," << wrapper.rectangle.width << ","
           << wrapper.rectangle.height << ")";
        return os;
    }
};

TEST_CASE("AlternativeFormattingCanBeEasyToRead")
{
    ApprovalTests::Approvals::verifyAll(
        "rectangles", getRectangles(), [](auto r, auto& os) {
            os << FormatRectangleForMultipleLines(r);
        });
}
```

(See snippet source)

Note The output operator (<<) needs to be declared before Approval Tests. Usually this is handled by putting it in its own header file, and including that at the top of the test source code.

3.5.5 Specialize StringMaker

If you want to use something other than an output operator (<<), one option is to create a specific specialization for StringMaker, for your specific type.

Here is an example:

```
template <>
std::string ApprovalTests::StringMaker::toString(const StringMakerPrintable& printable)
{
    return "From StringMaker: " + std::to_string(printable.field1_);
}
```

(See snippet source)

gcc 5 & 6

With older compilers, you might need to make the namespace explicit, like this:

```
namespace ApprovalTests
{
    template <> std::string StringMaker::toString(const StringMakerPrintable& printable)
    {
        return "From StringMaker: " + std::to_string(printable.field1_);
    }
}
```

(See snippet source)

3.5.6 Use `TApprovals<YourStringConvertingClass>`

If you want to change a broader category of how strings are created, you can create your own string-maker class, and tell Approvals to use it, using the template mechanism.

Here is how you create your own string-maker class:

```
class CustomToStringClass
{
public:
    template <typename T> static std::string toString(const T& printable)
    {
        return "From Template: " + std::to_string(printable.field1_);
    }
};
```

(See snippet source)

However, this alone will not do anything. You now need to call a variation of Approvals that uses it. You can do this directly by:

```
ApprovalTests::TApprovals<
    ApprovalTests::ToStringCompileTimeOptions<CustomToStringClass>>::verify(p);
```

(See snippet source)

Or you can create your own custom alias to use your customisation:

```
using MyApprovals = ApprovalTests::TApprovals<
    ApprovalTests::ToStringCompileTimeOptions<CustomToStringClass>>;

MyApprovals::verify(p);
```

(See snippet source)

With `MyApprovals::verify()`, we have not changed the behavior of `Approvals::verify()`.

If, instead, you want to change the default string formatting, so that all calls to `Approvals::verify()` and related methods will automatically use your new string formatter, you can override the default, by defining this macro **before including the Approval Tests header**.

```
#define APPROVAL_TESTS_DEFAULT_STREAM_CONVERTER StringMaker
```

(See snippet source)

3.5.7 See also

- Tips for Designing Strings
- How to Scrub Non-Deterministic Output
- How to Use the Fmt Library To Print Objects.

3.6 Tips for Designing Strings

When you use Approval tests, the results of the things you are testing are going to be stored on disk. It is good if you can diff the files, to gain an understanding of what is created and how they change. Mainly this is done by creating strings.

3.6.1 Design

If your code already has output operators, then go ahead and use them in Approvals.

If your code doesn't have output operators already, then here are some general guidelines to consider, to generate strings that work well with Approvals.

The general design rules when writing:

1. Objects print their relevant data
2. The data is consistent between runs (no times, pointers, random)
3. The data is easy to read

Note: for the same data, different tests might need different string conversions, to satisfy these rules.

Method	Example	Advantages	Disadvantages
XML	`<type > xml </type>`	Works with standard tools	Very verbose; hard to scan by eye
JSON	`{" type":"json"}`	Works with standard tools; less verbose	
YAML	`type:yaml`	Works with standard tools; less verbose	Indentation matters
simple	`( type: simple)`		It's a custom format
simpler	`(simpler)`		Does not include meta data
formatted	`(type )=(formatted)`	Works well for many lines of the same type of data, for example an array of rectangles	
tab-separated		Works with Excel and Markdown; works well for many lines of the same data	
co mma-separated	`type, csv`	Works with Excel	Works with Excel

3.6.2 Composability

TODO Explain things like:

- When are things very non-composable, e.g. hand-coded YAML

3.6.3 Lists

Some formats will be more readable when you are writing lists of objects. Here's an example of verifing a list of rectangles

```
ApprovalTests::Approvals::verifyAll("rectangles", getRectangles());
```

(See snippet source)

Notice how this:

```
rectangles


[0] = [x: 4 y: 50 width: 100 height: 61]
[1] = [x: 50 y: 5200 width: 400 height: 62]
[2] = [x: 60 y: 3 width: 7 height: 63]
```

(See snippet source)

compares to this:

```
rectangles


(x,y,width,height) = (4,50,100,61)
(x,y,width,height) = (50,5200,400,62)
(x,y,width,height) = (60,3,7,63)
```

(See snippet source)

3.6.4 Tools

TODO Explain things like:

- Using Excel to create graphs
- Loading run-time data from captured approval results
- Querying logs from JSON output
- IExecutable queries

3.6.5 How to Implement This

Approvals offers multiple ways to customise the To String. For details, see String conversions.

CHAPTER

FOUR

CUSTOMISING BEHAVIOUR

- **Principles**: *Options* | *Disposable Objects*
- **Customisation points**: *Reporters* | *Comparators* | *Writers* | *Namers* | *Scrubbers* | *Configuring Approval Tests*
- **Summary**: *All Customizations of Approval Tests*

4.1 Options

4.1.1 Introduction

There are many things you might want to tweak with Approval Tests. `Options` is the entry-point for many of the changes. It is on all `verify()` methods, as an optional parameter.

Note: If you are interested in why we moved from "optional Reporter arguments" to "optional Options arguments", see Why We Are Converting To Options.

4.1.2 Fluent Interface

Options utilizes a fluent interface, allowing you to chain together commands. Each returned object is a new copy.

```
ApprovalTests::Options()
    .withReporter(ApprovalTests::QuietReporter())
    .withScrubber(ApprovalTests::Scrubbers::scrubGuid)
    .fileOptions().withFileExtension(".json")
```

(See snippet source)

4.1.3 Reporters

Reporters launch diff tools upon failure. There are two ways to set a Reporter.

1. Pass in a Reporter object to the Options constructor, for example:

```
using namespace ApprovalTests;
Approvals::verify("text to be verified", Options(Windows::AraxisMergeReporter()));
```

(See snippet source)

2. Call `.withReporter()` on an existing Options object, for example:

```
using namespace ApprovalTests;
Approvals::verify("text to be verified",
                  Options().withReporter(Mac::AraxisMergeReporter()));
```

(See snippet source)

4.1.4 Scrubbers

Scrubbers clean output to help remove inconsistent pieces of text, such as dates. There are two ways to set a Scrubber.

1. Pass in a function pointer to the Options constructor, for example:

```
using namespace ApprovalTests;
Approvals::verifyAll("IDs", v, Options(Scrubbers::scrubGuid));
```

(See snippet source)

2. Call `.withScrubber()` with a function pointer, for example:

```
using namespace ApprovalTests;
Approvals::verifyAll("IDs", v, Options().withScrubber(Scrubbers::scrubGuid));
```

(See snippet source)

4.1.5 File Options

The `Options::FileOptions` class exists to customise the `.approved` and `.received` files in various ways.

For now, it just controls the file extension.

File Extensions

If you want to change the file extension of both the approved and received files, use `withFileExtension()`.

```
using namespace ApprovalTests;

Approvals::verify("text to be verified",
                  Options().fileOptions().withFileExtension(".xyz"));
```

(See snippet source)

Note: `withFileExtension()` returns an `Options` object, so it's possible to keep appending more `with...()` calls.

4.1.6 Defaults

The default constructor for Options does:

- no scrubbing
- uses file extension `.txt`
- uses whatever is currently set as *the default reporter*.

4.1.7 Adding to Existing Options object

Because of the fluent options, you can modify a specific option from an existing Options object, while retaining all other settings, and not changing the original object.

```
verifyWithQuietReporter(std::string text, const Options& o)
{
    Approvals::verify(text, o.withReporter(QuietReporter());
}
```

4.2 Disposable Objects

Most of the configuration points in Approval Tests return a "disposable object" (also known as RAII.) This means that when that object is deleted (when it goes out of scope), the customisation to the configuration is undone, and the previous configuration is restored.

This means it is important to store the result in a variable, to pay attention to the scope of that variable, so it lines up with how long you wish the configuration to exist.

All of our disposable objects take advantage of the C++17 language feature `[[nodiscard]]`. If you are on C++17, this means that the following line will either give a compiler warning or a compiler error, depending on your setup.

```
ApprovalTests::Approvals::useApprovalsSubdirectory("directory");
```

(See snippet source)

If you are on C++14 or below, the compiler will not detect this, but the code is still incorrect.

The code should look like this:

```
auto disposer = ApprovalTests::Approvals::useApprovalsSubdirectory("directory");
```

(See snippet source)

4.3 Reporters

4.3.1 Supported Diff Tools

The DiffReporter class goes through a chain of possible reporters to find the first option installed on your system. Currently the search goes in this order:

Mac

```
new AraxisMergeReporter(),
new BeyondCompareReporter(),
new DiffMergeReporter(),
new KaleidoscopeReporter(),
new P4MergeReporter(),
new SublimeMergeReporter(),
new KDiff3Reporter(),
new TkDiffReporter(),
```

(continues on next page)

(continued from previous page)

```
new VisualStudioCodeReporter(),
new CLionDiffReporter()
```

(See snippet source)

Linux

```
new BeyondCompareReporter(),
new MeldReporter(),
new SublimeMergeReporter(),
new KDiff3Reporter()
```

(See snippet source)

Windows

```
new TortoiseDiffReporter(), // Note that this uses Tortoise SVN Diff
new TortoiseGitDiffReporter(),
new BeyondCompareReporter(),
new WinMergeReporter(),
new AraxisMergeReporter(),
new CodeCompareReporter(),
new SublimeMergeReporter(),
new KDiff3Reporter(),
new VisualStudioCodeReporter(),
```

(See snippet source)

4.3.2 Cross-platform

These are all based on the diff tool being in the PATH.

```
new VisualStudioCodeReporter(),
```

(See snippet source)

4.3.3 Registering a default reporter

At present, the default Reporter is the DiffReporter. Whenever you call Approvals, you have the chance to pass in your own Reporter. However, if you would like to change what the default reporter is when you don't pass in a specific Reporter, you can do this at a global or per-test level, by adding the line:

```
// main.cpp:
#include <memory>
auto defaultReporterDisposer = ApprovalTests::Approvals::useAsDefaultReporter(
    std::make_shared<ApprovalTests::DiffReporter>());
```

(See snippet source)

The return value is "Disposable", meaning it will restore the original reporter when the object destructs. Because of this, if you do not store the result in a variable, it will immediately undo itself by the end of the line.

4.3.4 Front Loaded Reporters

By default, Approval tests will not launch any reporters on supported CI machines. To do this, we use front loaded reporters…

Front loaded reporters allow you to block all normal reporting behaviour. This is useful in situations like running on a CI Machine, where you wouldn't want a reporter to open.

For more information, see Build Machines and Continuous Integration servers.

Here is an example of not launching any reporters if you are on a machine with a particular name, by using BlockingReporter.

```
// main.cpp
auto frontLoadedReportDisposer = ApprovalTests::Approvals::useAsFrontLoadedReporter(
    ApprovalTests::BlockingReporter::onMachineNamed("MyCIMachineName"));
```

(See snippet source)

Once you have added that, even calling approvals with a specific Reporter will not launch it on the CI system (but will for all other systems). For example:

```
using namespace ApprovalTests;
Approvals::verify("text to be verified", Options(Windows::AraxisMergeReporter()));
```

(See snippet source)

Blocking Reporters

Blocking reporters are a simple class, designed for use with FrontLoadedReporters, to prevent launching of reporters in certain environments.

```
// main.cpp
auto frontLoadedReportDisposer = ApprovalTests::Approvals::useAsFrontLoadedReporter(
    ApprovalTests::BlockingReporter::onMachineNamed("MyCIMachineName"));
```

(See snippet source)

4.3.5 Miscellaneous Helper Reporters

While most reporters open some sort of external program, for the purpose of understanding how the tests went wrong, and verifying the correct answer, there are some reporters that are helpful for specific situations.

Auto-approving

There are three reporters that can help with the approving of single or multiple tests.

- `AutoApproveIfMissingReporter`: if there is no approved file already, the received file will automatically be copied over the approved one. Otherwise, it does nothing. One possible cause for confusion here is if you ran the test previously with a standard reporter, that will have created an almost-empty approved file, which will then block this from working.
- `ClipboardReporter`: this puts the command-line to moved the approve file on to your computer's clipboard. You then review this, and paste it in to a terminal window. This only works with one test at a time.
- `AutoApproveReporter`: be careful, this will overwrite every existing ".approved" file, with no confirmation. This is best used when you are expecting large numbers of files that are already version-controlled to be updated, and you would rather review the changes in your version control system.

4.3.6 Related Pages

- How to Use A Custom Reporter
- How to Select a Reporter with an Environment Variable
- How to Submit a New Reporter to ApprovalTests

4.4 Custom Comparators

4.4.1 Default Behaviour

When Approval Tests compares the "expected" and "approved" files to verify that they are the same, it reads both files, one character at a time, and tests if their contents are identical.

Note that it **ignores differences in line endings**, to ensure that files written on Windows will match thos written on Unix, and vice versa.

In almost all uses of Approval Tests, this behaviour works well.

4.4.2 Registering a custom comparator

Overview

This section shows how to customise the way that file contents are compared in Approval Tests.

Real-world example scenario:

- Suppose you are approving image files, and for some reason, there are sometimes tiny differences in the red, green or blue colour values. Or perhaps on some test environments, the images are written 24-bit files, and on others they are 32-bit.
- You know that no human would ever be able to detect these tiny differences visually, so you want Approval Tests to treat files with only these tiny differences as equivalent.
- But Approval Tests doesn't know this: it just sees a 24-bit .png file and a 32-bit one as having different content.
- So you wish to customise Approval Tests to read the colour values in the two files, and ignore any tiny differences in these values.

There are two steps needed to get Approval Tests to use a custom file comparison:

1. A custom implementation of the `ApprovalComparator` interface, that reads two files and reports if they are equivalent
2. A call to `FileApprover::registerComparatorForExtension()`, supplying an instance of the custom comparator, and the file extension it should be used for.

Example Code

In this trivial example, we want to treat two files as matching if their contents are the same length.

First, we write our custom `ApprovalComparator` implementation:

```
class LengthComparator : public ApprovalTests::ApprovalComparator
{
public:
    bool contentsAreEquivalent(std::string receivedPath,
                               std::string approvedPath) const override
    {
        return ApprovalTests::FileUtils::fileSize(receivedPath) ==
               ApprovalTests::FileUtils::fileSize(approvedPath);
    }
};
```

(See snippet source)

Then we call `FileApprover::registerComparatorForExtension()` to tell Approval Tests to use `LengthComparator` to compare all files with extension `.length`. This customisation will last for the rest of the test run, and we would typically put this in our `main.cpp`.

```
auto disposer = FileApprover::registerComparatorForExtension(
    ".length", std::make_shared<LengthComparator>());
```

(See snippet source)

The return value is "Disposable", meaning it will restore the original comparator when the object destructs. Because of this, if you do not store the result in a variable, it will immediately undo itself by the end of the line.

4.5 Writers

4.5.1 Default Behaviour

By default, when Approval Tests verifies an object, it uses the StringWriter class to write out the string representation of the object.

And by default, `StringWriter` gives the saved file the extension `.txt`.

`StringWriter` implements the ApprovalWriter interface.

4.5.2 Using custom writers

Suppose that you are serialising an object that cannot easily be represented in a text file, such as an image. In this case, the built-in `StringWriter` is not suitable, and you will need to write and use a custom implementation of `ApprovalWriter`.

Here is a simple example of using a custom writer to produce an HTML file.

```
HtmlWriter writer("<h1>hello world</h1>", ".html");
ApprovalTests::Approvals::verify(writer);
```

(See snippet source)

4.5.3 Using custom filename extensions

Suppose that you are serializing an object in some text format like `JSON` or `CSV`. By writing to this file extension, different tools will render it appropriately.

If all you want to do is change the file extension, here is how:

```
ApprovalTests::Approvals::verify(
    "<h1>hello world</h1>",
    ApprovalTests::Options().fileOptions().withFileExtension(".html"));
```

(See snippet source)

4.5.4 Empty Files

Most reporters will create a `.approved.` file if one does not exist, as most diff tools do not handle a missing file well. By default, these empty files are empty text files with an empty string `""`.

If the file extension is not a text file (for example, a PNG), you will still get this behaviour, which can result in some diff tools saying 'this is not a valid file'.

ApprovalTests allows for you to customize this behaviour.

For a tool which will help you with this, see EmptyFiles.

Customizing by File Extension

To add new behaviour for a specific file extension, you can register customizations as follows:

```
ApprovalTests::EmptyFileCreator htmlCreator = [](std::string fileName) {
    ApprovalTests::StringWriter s("<!doctype html><title>TITLE</title>");
    s.write(fileName);
};
ApprovalTests::EmptyFileCreatorByType::registerCreator(".html", htmlCreator);
```

(See snippet source)

This will leave the creation of files with all other extensions alone, and will last for the remainder of program execution, or until a new creator for this file extension is registered.

Full Customization

Here is an example of replacing the entire empty file creation code, for the lifetime of the disposer.

New `.approved.` files will be created with minimal, valid HTML content, and everything else will be empty.

```
ApprovalTests::EmptyFileCreator htmlCreator = [](std::string fileName) {
    std::string contents = "";
    if (ApprovalTests::StringUtils::endsWith(fileName, ".html"))
    {
        contents = "<!doctype html><title>TITLE</title>";
    }
    ApprovalTests::StringWriter s(contents);
    s.write(fileName);
};
auto disposer = ApprovalTests::FileUtils::useEmptyFileCreator(htmlCreator);
```

(See snippet source)

4.6 Namers

4.6.1 The Purpose of Namers

`Approvals::verify(text);`

could be written as:

`REQUIRE(text == (loadContentsFromFile("FileName.TestName.approved.txt"));`

Part of Approval Tests' "Convention over Configuration" is to remove this by automatically creating meaningful file names, and therefore it could be written as:

`REQUIRE(text == (loadContentsFromFile(namer.getApprovedFile()));`

"If you **always** have to do something, you should **never** have to do something."

Since all of your `REQUIRE`s would look like this, we can simplify it with the above `Approvals::verify(text);` - and this is enabled by the ApprovalNamers.

4.6.2 The Parts of Namers

The conventional layout for files saved with `Approvals::verify()` and related functions is:

- `path_to_test_file/FileName.TestName.approved.txt`
- `path_to_test_file/FileName.TestName.received.txt`

The Approval Namer is responsible for creating these two names.

The interface for this is ApprovalNamer.

Converting Test Names to Valid FileNames

Some C++ test frameworks allow test names to contain characters that are not valid in file or directory names on every operating system.

Therefore, by default, ApprovalTests will convert any non-valid filename character to an _ (underscore). This can mean `"test <"` and `"test >"` would produce colliding `test__.approved.txt`.

This behavior is customizable, here's an example:

```
TEST_CASE("Sanitizer <3 fileNames")
{
    {
        auto disposer =
            ApprovalTests::Approvals::useFileNameSanitizer([](std::string incoming) {
                return ApprovalTests::StringUtils::replaceAll(
                    incoming, " <3 ", "_loves_");
            });
```

(See snippet source)

4.6.3 Registering a Custom Namer

Locally

If you want to use a specific namer for a specific test, the easiest way is via Options:

```
auto namer = ApprovalTests::TemplatedCustomNamer::create(
    "{TestSourceDirectory}/{ApprovalsSubdirectory}/CustomName.{ApprovedOrReceived}.
↪{FileExtension}");
ApprovalTests::Approvals::verify("Hello", ApprovalTests::Options().withNamer(namer));
```

(See snippet source)

Globally

If you ever want to create a custom namer, that's used in multiple places, Approval Tests has a mechanism to change which namer it uses by default. Please note that you need to create a function that creates new namers.

```
auto default_namer_disposer = ApprovalTests::Approvals::useAsDefaultNamer(
    []() { return std::make_shared<FakeNamer>(); });
```

(See snippet source)

Hint: Many namer classes have a `useAsDefaultNamer()` convenience method to do this for you.

4.6.4 Alternative Namers

TemplatedCustomNamer

The easiest way to create a custom namer is to use a `TemplatedCustomNamer`.

As well as giving great flexibility, this introduces the ability to run Approval Tests on machines that do not have the source code, such as when doing cross-compilation

Here is an example:

```
ApprovalTests::TemplatedCustomNamer namer(
    "/my/source/directory/{ApprovedOrReceived}/"
    "{TestFileName}.{TestCaseName}.{FileExtension}");
```

(See snippet source)

Note: The character / will be converted to \ on Windows machines, at run-time.

Supported tags

```
auto testSourceDirectory = "{TestSourceDirectory}";
auto relativeTestSourceDirectory = "{RelativeTestSourceDirectory}";
auto approvalsSubdirectory = "{ApprovalsSubdirectory}";
auto testFileName = "{TestFileName}";
auto testCaseName = "{TestCaseName}";
auto approvedOrReceived = "{ApprovedOrReceived}";
auto fileExtension = "{FileExtension}";
```

(See snippet source)

Here is some output showing examples with these tags expanded:

```
For template: {RelativeTestSourceDirectory}/{ApprovalsSubdirectory}/{TestFileName}.
↪{TestCaseName}.{ApprovedOrReceived}.{FileExtension}

Result: namers/approval_tests/TemplatedCustomNamerTests.Demo_all_namer_templates.
↪approved.txt

With breakdown:
RelativeTestSourceDirectory = namers/
ApprovalsSubdirectory       = approval_tests/
TestFileName                = TemplatedCustomNamerTests
TestCaseName                = Demo_all_namer_templates
ApprovedOrReceived          = approved
FileExtension               = txt

Also available:
{TestSourceDirectory} = <full path to sources>/ApprovalTests.cpp/tests/DocTest_Tests/
↪namers/
```

(See snippet source)

Examples

If you would like to see an example of this running for scenarios where the execution is in a separate environment from the compilation, check out our out_of_source example.

SeparateApprovedAndReceivedDirectoriesNamer

The pattern used by this class for file names is:

```
auto path = "{TestSourceDirectory}/{ApprovalsSubdirectory}/{ApprovedOrReceived}/
↪{TestFileName}.{TestCaseName}.{FileExtension}";
```

(See snippet source)

Which results in these file names:

- `./approved/{TestFileName}.{TestCaseName}.{FileExtension}`
- `./received/{TestFileName}.{TestCaseName}.{FileExtension}`

This layout enables you to use Beyond Compare 4 (or any other directory comparison tool) to do a folder/directory comparison, in order to compare pairs of files in the `approved/` and `received/` directories, and approve one or more files by copying them (without renaming) from `received/` to `approved/`.

The `approved/` and `received/` directories are created automatically.

To register this as your default namer, use:

```
auto default_namer_disposer =
    ApprovalTests::SeparateApprovedAndReceivedDirectoriesNamer::useAsDefaultNamer();
```

(See snippet source)

When using this namer, you will want to add the following line to your `.gitignore` file:

```
**/received/
```

4.6.5 Approving multiple files from one test

See MultipleOutputFilesPerTest.

4.7 Scrubbers

4.7.1 Introduction

If you are having trouble getting tests running reproducibly, you might need to use a “scrubber” to convert the non-deterministic text to something stable.

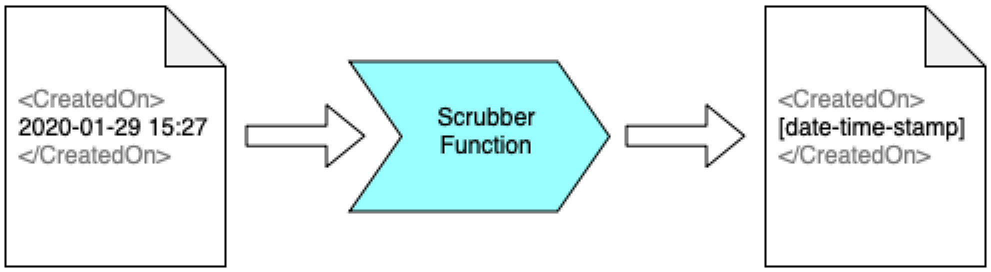


Fig. 1: Scrubber Overview

4.7.2 Interface

Fundamentally, a scrubber is function that takes a string and returns a string. You can create ones by passing in a function or a lambda. We also have some pre-made ones for your convenience.

4.7.3 How to use Scrubbers

You can scrub text manually, before passing it in to Approvals::verify(), but the preferred method is to include a Scrubber as an option.

```
ApprovalTests::Approvals::verify(input,
                                 ApprovalTests::Options().withScrubber(scrubber));
```

(See snippet source)

4.7.4 Scrubber concepts

There are several patterns that are commonly used when scrubbing:

- Replace troublesome text
- Replace troublesome text, tracking duplicates
- Combining scrubbers
- Deleting troublesome lines

4.7.5 See also

- How to Scrub Non-Deterministic Output

4.8 Configuring Approval Tests

4.8.1 Using sub-directories for approved files

If you have a lot of approval files, you might want to put them in a subdirectory, to prevent them cluttering up your source files. You can do this at a global or per-test level, by adding the line:

```
auto directoryDisposer =
    ApprovalTests::Approvals::useApprovalsSubdirectory("approval_tests");
```

(See snippet source)

Note that the sub-directory is created automatically, and that it will be inside the directory containing the source code of the test, not the current working directory of the test process.

The return value will restore the original directory when the object destructs. Because of this, if you do not store the result in a variable, it will immediately undo itself by the end of the line.

This mechanism allows you to select a different sub-directory in individual tests.

4.9 All Customizations of Approval Tests

4.9.1 Introduction

This page provides a starting point to see all the ways of controlling the behaviour of Approval Tests.

4.9.2 Disposable Objects

See Disposable Objects for a general overview of how most customisation of Approval Tests works.

4.9.3 Namers - how Approved Files are named

Output Directories

See Using sub-directories for approved files for putting Approval files in a sub-directory, to keep your tests directory tidy.

See SeparateApprovedAndReceivedDirectoriesNamer to put the Approved and Received files in adjacent directories, in order to allow a folder-diffing tool to compare the two directories.

Output File Names

See Namers for controlling how Approval files are named.

See Multiple output files per test for multiple ways to control file names when verifying more than one output file in a test.

See Using Approval Tests With Google Tests for how to control the names of Approval files if using that framework.

Filename Extensions: Although it seems like the file extension would be created by the namer, it is actually created by the writer. See below.

4.9.4 Writers - how things being verified are written to file

See Writers for how to customize the serialization of objects.

Filename Extensions: If you only want to change a file extension, see Using custom filename extensions.

4.9.5 Comparators - how files are compared

See Custom Comparators for controlling how Approval Tests determines if the Received File and Approved File are equivalent.

4.9.6 Reporters - how test failures are shown

See Reporters to control how test failures are shown. This is typically done by showing a diff tool.

4.9.7 Scrubbers - non-deterministic output

See Scrubbers for various ways to deal with non-deterministic output.

CHAPTER

FIVE

COMMON CHALLENGES

- **Challenges**: *Multiple output files per test*

5.1 Multiple output files per test

5.1.1 Introduction

ApprovalTests uses the name of the current test to determine the names of output files that it writes. This means that **by default, there is only one approval file per test case**.

However, sometimes it is useful to be able to verify multiple files in one test case, or have a file per OS or other environment configuration.

5.1.2 Scenarios

Here are some examples of files you might want.

Multiple data inputs:

In this scenario, your test creates 3 files, all of which are being checked when you run the test.

```
TestProteinGeneration.createImage.protein1.approved.png
TestProteinGeneration.createImage.protein2.approved.png
TestProteinGeneration.createImage.protein3.approved.png
```

Multiple outputs:

In this scenario, the code under test creates three different types of files, all of which are being checked.

```
TestProtein.processInput.logOutput.approved.txt
TestProtein.processInput.calculationResults.approved.txt
TestProtein.processInput.renderedResult.approved.png
```

Multiple environments:

In this scenario, your test only creates one file, and which one it is checked against is dependent on which OS the test is running on.

```
TestQtDialog.loginScreen.onMacOSX.approved.png
TestQtDialog.loginScreen.onWindows.approved.png
TestQtDialog.loginScreen.onLinux.approved.png
```

5.1.3 Mechanisms

Here are a few ways to do that.

Catch2

You can have a file-per-subsection.

You can either do these dynamically, e.g. in a for-loop:

```
TEST_CASE("MultipleOutputFiles-DataDriven")
{
    // This is an example of how to write multiple different files in a single test.
    // Note: For data as small as this, in practice we would recommend passing the
    // greetings container in to Approvals::verifyAll(), with a lambda to format the␣
↪output,
    // in order to write all data to a single file.
    std::vector<Greeting> greetings{
        Greeting(British), Greeting(American), Greeting(French)};
    for (auto greeting : greetings)
    {
        SECTION(greeting.getNationality())
        {
            ApprovalTests::Approvals::verify(greeting.getGreeting());
        }
    }
}
```

(See snippet source)

Or hard-coded, with multiple sections:

```
TEST_CASE("MultipleOutputFiles-ForOneObject")
{
    Greeting object_under_test;
    SECTION("British")
    {
        ApprovalTests::Approvals::verify(object_under_test.getGreetingFor(British));
    }
    SECTION("American")
    {
        ApprovalTests::Approvals::verify(object_under_test.getGreetingFor(American));
    }
    SECTION("French")
    {
        ApprovalTests::Approvals::verify(object_under_test.getGreetingFor(French));
    }
}
```

(See snippet source)

Note: Catch2 sub-sections continue to run even if the previous one failed. This is useful, as it allows you to approve all the files in one test run.

doctest

You can have a file-per-subcase.

Note: unlike Catch, doctest sub-cases must have static strings for names, so if you want to name things dynamically, you will have to use the native Approval Tests mechanism - see below.

You can have hard-coded, with multiple sections:

```
TEST_CASE("MultipleOutputFiles-ForOneObject")
{
    using namespace ApprovalTests;

    Greeting object_under_test;
    SUBCASE("British")
    {
        Approvals::verify(object_under_test.getGreetingFor(British));
    }
    SUBCASE("American")
    {
        Approvals::verify(object_under_test.getGreetingFor(American));
    }
    SUBCASE("French")
    {
        Approvals::verify(object_under_test.getGreetingFor(French));
    }
}
```

(See snippet source)

Approval Tests

Approval Tests also allows for multiple files per test, via the `NamerFactory`. This works for all supported test frameworks.

You can either do these dynamically, e.g. in a for-loop:

```
TEST_CASE("ApprovalTests-MultipleOutputFiles-DataDriven")
{
    using namespace ApprovalTests;

    // This is an example of how to write multiple different files in a single test.
    // Note: For data as small as this, in practice we would recommend passing the
    // greetings container in to Approvals::verifyAll(), with a lambda to format the␣
↪output,
    // in order to write all data to a single file.
    std::vector<Greeting> greetings{
        Greeting(British), Greeting(American), Greeting(French)};
    for (auto greeting : greetings)
    {
        auto section = NamerFactory::appendToOutputFilename(greeting.getNationality());
        Approvals::verify(greeting.getGreeting());
    }
}
```

(See snippet source)

Or hard-coded, with multiple sections:

```
TEST_CASE("ApprovalTests-MultipleOutputFiles-ForOneObject")
{
    using namespace ApprovalTests;

    Greeting object_under_test;
    {
        auto section = NamerFactory::appendToOutputFilename("British");
        Approvals::verify(object_under_test.getGreetingFor(British));
    }
    {
        auto section = NamerFactory::appendToOutputFilename("American");
        Approvals::verify(object_under_test.getGreetingFor(American));
    }
    {
        auto section = NamerFactory::appendToOutputFilename("French");
        Approvals::verify(object_under_test.getGreetingFor(French));
    }
}
```

(See snippet source)

5.1.4 Approving multiple files in one test

Often, when you have multiple output files in one test, there is an annoyance that you have to run the test multiple times, once per output file. This is because Approvals throws an exception after each test failure, which the test framework immediately traps as a failure, stopping your test from writing the remaining files.

There are two things that work well to do this.

First, you can use an ExceptionCollector so that the test does not stop after the first failure.

Second, you can use AutoApproveIfMissingReporter so that the first time a verify is run, it is automatically approving the result. There are more options in Auto-approving

For example:

```
TEST_CASE("ApprovalTests-MultipleOutputFiles-AutoApprove")
{
    using namespace ApprovalTests;

    ExceptionCollector exceptions; // Allow all files to be written, regardless of errors
    std::vector<Greeting> greetings{
        Greeting(British), Greeting(American), Greeting(French)};
    for (auto greeting : greetings)
    {
        auto section = NamerFactory::appendToOutputFilename(greeting.getNationality());
        exceptions.gather([&]() {
            Approvals::verify(
                greeting.getGreeting(),
                Options(
                    AutoApproveIfMissingReporter())); // Automatically approve first time
```

(continues on next page)

(continued from previous page)

```
        });
    }
    exceptions.release();
}
```

(See snippet source)

CHAPTER

SIX

HOW-TO GUIDES

6.1 How to Test the Contents of a Container

6.1.1 When to test a container

You have called a method that returns a collection of objects, and you want a nicer read-out than the standard verify() will give you.

6.1.2 Steps

1. Copy this starter text.

```
std::vector<std::string> objectsToVerify{"hello", "world"};
ApprovalTests::Approvals::verifyAll("TITLE", objectsToVerify);
```

(See snippet source)

which would produce:

```
TITLE


```

(continues on next page)

(continued from previous page)

```
[0] = hello
[1] = world
```

(See snippet source)

2. Replace the `objectsToVerify` container with your collection of objects.
3. Change the TITLE to something meaningful
4. Run it, and approve the output.

6.1.3 More Examples

For some examples, see VectorTests.cpp.

6.1.4 Further Advice

For advice on effective formatting, see Tips for Designing Strings. As you write out larger volumes of data in your approval files, experience has shown that the choice of layout of text in approval files can make a big difference to maintainability of tests, when failures occur.

6.2 How to Test a Variety of Values for One Input

6.2.1 When to use Approvals::verifyAll()

When you want to test a lot of variations for a single input value.

If you have more than parameter that you want to vary, check out Testing Combinations.

6.2.2 Steps

1. Copy this starter text.

```
TEST_CASE("VerifyAllStartingPoint")
{
    std::vector<std::string> inputs{"input.value1", "input.value2"};
    ApprovalTests::Approvals::verifyAll("TITLE", inputs, [](auto input, auto& stream) {
        stream << input << " => "
               << "placeholder";
    });
}
```

(See snippet source)

2. Modify the input container for your chosen values.
3. Run it, and make sure that you have your inputs wired up correctly.

If they are wired up correctly, you will see a file that looks like this: it is the left hand side of the file that matters at this point: all combinations of your own input values should be listed:

```
TITLE


input.value1 => placeholder
input.value2 => placeholder
```

(See snippet source)

4. Replace the "placeholder" with a call to the functionality that you want to test.
5. Change the TITLE to something meaningful
6. Run it, and approve the output.

6.2.3 Further Advice

For advice on effective formatting, see Tips for Designing Strings. As you write out larger volumes of data in your approval files, experience has shown that the choice of layout of text in approval files can make a big difference to maintainability of tests, when failures occur.

6.3 How to Scrub Non-Deterministic Output

6.3.1 Scrubbers

This page assumes that you understand the concept of Scrubbers.

6.3.2 Lambda example

```
using namespace ApprovalTests;
Approvals::verify(
    "1 2 3 4 5 6",
    Options().withScrubber(
        [](const std::string& t) {return StringUtils::replaceAll(t, "3", "Fizz");}
    ));
```

(See snippet source)

This would produce:

```
1 2 Fizz 4 5 6
```

(See snippet source)

(In the real world, scrubbers are generally used to remove text that is expected to differ between test runs… Here, we use a trivial example for ease of explanation.)

6.3.3 Pre-made Scrubbers

Regular Expressions (regex)

API for scrubbing with regex

Approval Tests provides lots of convenience methods to scrub text based on regular expressions.

Using a regex search term

For example, here is an example where random numbers are scrubbed:

```
using namespace ApprovalTests;
std::stringstream os;
os << "Hello " << random(1000) << " World";
Approvals::verify(os.str(),
                  Options(Scrubbers::createRegexScrubber(R"(\d+)", "[number]")));
```

(See snippet source)

This will produce:

```
Hello number World
```

(See snippet source)

Note: In the above example, the caller passes in a `std::string`, and for convenience of the calling code, Approval Tests converts that to a `std::regex`. The calling code is responsible for making sure that the string contains a valid regular expression.

Using a lambda for greater control of replacement text

There are many combinations of these parameters, that allow for customization at whatever level you need, the most complex being:

```
auto input = "1) Hello 1234 World";
auto scrubber = ApprovalTests::Scrubbers::createRegexScrubber(
    std::regex(R"(\d+)"), [](const auto& match) {
        auto match_text = match.str();
        auto match_integer = std::stoi(match_text);
        if (match_integer < 10)
        {
            return match_text;
        }
        else
        {
            return std::string("[number]");
        }
    });
```

(See snippet source)

This will produce:

```
1) Hello [number] World
```

(See snippet source)

See also

- API Reference for Scrubbers
- Regex Tutorial
- Regex Cheat Sheet
- Regex Testing Tool
- C++ Regex Reference

GUID

You can scrub GUIDs by using a pointer to the function `Scrubbers::scrubGuid`.

For example the following code:

```
using namespace ApprovalTests;
std::string jsonFromRestCall = R"(
    {
        child: {
            id: b34b4da8-090e-49d8-bd35-7e79f633a2ea
            parent1: 2fd78d4a-ad49-447d-96a8-deda585a9aa5
            parent2: 05f77de3-3790-4d45-b045-def96c9cd371
        }
        person: {
            name: mom
            id: 2fd78d4a-ad49-447d-96a8-deda585a9aa5
        }
        person: {
            name: dad
            id: 05f77de3-3790-4d45-b045-def96c9cd371
        }
    }
    )";
Approvals::verify(jsonFromRestCall, Options().withScrubber(Scrubbers::scrubGuid));
```

(See snippet source)

will produce:

```
{
            child: {
                id: guid_1
                parent1: guid_2
                parent2: guid_3
            }
            person: {
                name: mom
```

(continues on next page)

(continued from previous page)

```
            id: guid_2
        }
        person: {
            name: dad
            id: guid_3
        }
    }
```

(See snippet source)

Notice that when GUIDs are repeated within the same file, they are replaced with the same text.

6.4 How to Use the Fmt Library To Print Objects

6.4.1 Introduction

{fmt} is a useful library for printing objects with many default types formatted out of the box.

6.4.2 Usage

Simply use `FmtApprovals::`

For example, vectors are not `ostream (<<)` printable by default. However, they are with {fmt}. so :

```
std::vector<int> numbers = {1, 2, 3};
ApprovalTests::FmtApprovals::verify(numbers);
```

(See snippet source)

This will produce the following output:

```
{1, 2, 3}
```

(See snippet source)

note: it is important that we included fmt before approvaltests.

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

(See snippet source)

6.4.3 Installation

Bring your own

ApprovalTests assumes you will add the {fmt} library **before** including `ApprovalTests.hpp`. As such it makes no assumptions about fmt. We suggest you read their docs.

If you would like to see how we added fmt to our build, check out:

```
target_link_libraries(${PROJECT_NAME} fmt::fmt)
```

(See snippet source)

fmt/CmakeList.txt

fmt/CmakeList.txt.in

Set as default for Approvals

If you wish, you can set FmtApprovals to be the default Approvals with the following line before including `ApprovalTests.hpp`

```
#define APPROVAL_TESTS_DEFAULT_STREAM_CONVERTER FmtToString
```

(See snippet source)

6.5 How to Use A Custom Reporter

This guide is for creating the ability to use a custom reporter that works on your machine.

For figuring out how to make a more robust custom reporter, that you might want to submit back to us as a Pull Request, check out How to Submit a New Reporter to ApprovalTests.

Let's say that you really enjoy using Sublime Merge, and on your system it's located in `"/Applications/Sublime Merge.app/Contents/SharedSupport/bin/smerge"`

If you were to run this the command line, the full command would look this this:

```
"/Applications/Sublime Merge.app/Contents/SharedSupport/bin/smerge" mergetool --no-wait
↪"test.received.txt" "test.approved.txt" -o "test.approved.txt" &
```

(See snippet source)

You can do this simply by creating a Reporter using:

```
auto path = "/Applications/Sublime Merge.app/Contents/SharedSupport/bin/smerge";
auto arguments = "mergetool --no-wait {Received} {Approved} -o {Approved}";
auto reporter = ApprovalTests::CustomReporter::create(path, arguments);
```

(See snippet source)

By default, this will run in the background. Most of the time this is what you want.

However, you can force it to run in the foreground with:

```
auto reporter =
    ApprovalTests::CustomReporter::createForegroundReporter(path, arguments);
```

(See snippet source)

On Windows, you can specify a search path for the installed location of a program with `{ProgramFiles}`.

```
auto path = "{ProgramFiles}Beyond Compare 4\\BCompare.exe";
auto reporter = ApprovalTests::CustomReporter::create(path);
```

(See snippet source)

See Registering a default reporter for wiring up this reporter as default, or you can dereference it and pass it in to individual `verify("text", *reporter)` method calls…

6.6 How to Select a Reporter with an Environment Variable

6.6.1 Introduction

This guide is for selecting a Reporter at run-time, by setting the value of an Environment Variable.

Why

Sometimes different members of your team will want to use different diff programs for reporting. This feature allows everyone to have a different choice.

Alternatively, sometimes you might want to view the results of a failing test result in a different diff tool, without having to recompile your code.

Warning!

Please don't use this as your first option, as it can add inconsistency with your build and someone else's. For example, if you are working alone on code, you can just set your reporter in code.

It's generally considered poor practice to use environment variables to control the behaviour of software builds and execution.

6.6.2 Setting the Environment Variable

The environment value `APPROVAL_TESTS_USE_REPORTER` can be set, to select the diff tool to use to show differences, if an Approval Test fails.

For example, to set Araxis Merge as your reporter, set the `APPROVAL_TESTS_USE_REPORTER` environment value to `AraxisMerge`.

6.6.3 Available Values

The following values of `APPROVAL_TESTS_USE_REPORTER` are currently recognised. See the next section for alternative variations on these names.

```
AutoApproveIfMissingReporter
AutoApproveReporter
CIBuildOnlyReporter
ClipboardReporter
CrossPlatform::VisualStudioCodeReporter
DefaultFrontLoadedReporter
DefaultReporter
DiffReporter
EnvironmentVariableReporter
Linux::BeyondCompareReporter
Linux::KDiff3Reporter
Linux::MeldReporter
Linux::SublimeMergeReporter
Mac::AraxisMergeReporter
Mac::BeyondCompareReporter
Mac::CLionDiffReporter
```

(continues on next page)

(continued from previous page)

```
Mac::DiffMergeReporter
Mac::KDiff3Reporter
Mac::KaleidoscopeReporter
Mac::P4MergeReporter
Mac::SublimeMergeReporter
Mac::TkDiffReporter
Mac::VisualStudioCodeReporter
QuietReporter
TextDiffReporter
Windows::AraxisMergeReporter
Windows::BeyondCompareReporter
Windows::CodeCompareReporter
Windows::KDiff3Reporter
Windows::SublimeMergeReporter
Windows::TortoiseDiffReporter
Windows::TortoiseGitDiffReporter
Windows::VisualStudioCodeReporter
Windows::WinMergeReporter
```

(See snippet source)

6.6.4 Acceptable Variations

A fully qualified name is:

`Windows::AraxisMergerReporter`

This breaks down in to:

`{Windows::}{AraxisMerger}{Reporter}`

or:

`{OS::}{Program}{Reporter}`

1. `{OS::}`
   - This is optional: it is one of `Mac::`, `Linux::` and `Windows::` and will be inferred if left out
   - Some reporters, such as `AutoApproveReporter`, don't have this component.
2. `{Program}`
   - This is **required**.
3. `{Reporter}`
   - This is optional, and is added if missing.

So, for example, on Windows, you can select `Windows::AraxisMergerReporter` by setting `APPROVAL_TESTS_USE_REPORTER` to any of the following values:

- `Windows::AraxisMergerReporter`
- `Windows::AraxisMerge`
- `AraxisMergerReporter`
- `AraxisMerge`

6.6.5 Other Information

This environment value is ignored on Continuous Integration builds.

6.7 How to Submit a New Reporter to ApprovalTests

This guide is for figuring out how to make a more robust custom reporter, that you might want to submit back to us as a Pull Request.

For creating the ability to use a custom reporter that works on your machine, see How to Use A Custom Reporter

6.7.1 Adding a new Mac reporter

By way of an example, for supporting a new Reporter on macOS, the steps are:

Implement the reporter

Edit ApprovalTests/reporters/DiffPrograms.h

- Add a declaration for the new function to the `Mac` namespace.
- If you are adding a tool that is already supported on an existing platform, please try to be consistent with naming.

```
DiffInfo ARAXIS_MERGE();
```

(See snippet source)

Edit ApprovalTests/reporters/DiffPrograms.cpp

- Add a new `APPROVAL_TESTS_MACROS_ENTRY` value to the `Mac` namespace, to create the definition for the new function.

```
APPROVAL_TESTS_MACROS_ENTRY(
    ARAXIS_MERGE,
    DiffInfo("/Applications/Araxis Merge.app/Contents/Utilities/compare",
             Type::TEXT_AND_IMAGE))
```

(See snippet source)

Edit ApprovalTests/reporters/MacReporters.h

- Add a declaration for the new reporter.
- In the most common case, this will be a new implementation of `GenericDiffReporter`

```
class AraxisMergeReporter : public GenericDiffReporter
{
public:
    AraxisMergeReporter();
};
```

(See snippet source)

Edit ApprovalTests/reporters/MacReporters.cpp

- Add a definition for the new reporter.
- This will use the `APPROVAL_TESTS_MACROS_ENTRY` you added in the first step.

```
AraxisMergeReporter::AraxisMergeReporter()
    : GenericDiffReporter(DiffPrograms::Mac::ARAXIS_MERGE())
{
}
```

(See snippet source)

- Scroll to the end of this file, and add an instance of the new reporter class to the `MacDiffReporter`
  - The reporters are searched in order, so more commonly-used or highly-featured ones should go nearer the start.
  - Paid-for ones should go before free ones.

```
new AraxisMergeReporter(),
new BeyondCompareReporter(),
new DiffMergeReporter(),
new KaleidoscopeReporter(),
new P4MergeReporter(),
new SublimeMergeReporter(),
new KDiff3Reporter(),
new TkDiffReporter(),
new VisualStudioCodeReporter(),
new CLionDiffReporter()
```

(See snippet source)

Other files to edit

Edit ApprovalTests/reporters/ReporterFactory.cpp

- Add a new `APPROVAL_TESTS_REGISTER_REPORTER` line, for your reporter class.

```
APPROVAL_TESTS_REGISTER_REPORTER(Mac::AraxisMergeReporter);
APPROVAL_TESTS_REGISTER_REPORTER(Mac::BeyondCompareReporter);
APPROVAL_TESTS_REGISTER_REPORTER(Mac::DiffMergeReporter);
APPROVAL_TESTS_REGISTER_REPORTER(Mac::KaleidoscopeReporter);
APPROVAL_TESTS_REGISTER_REPORTER(Mac::P4MergeReporter);
APPROVAL_TESTS_REGISTER_REPORTER(Mac::SublimeMergeReporter);
APPROVAL_TESTS_REGISTER_REPORTER(Mac::KDiff3Reporter);
APPROVAL_TESTS_REGISTER_REPORTER(Mac::TkDiffReporter);
APPROVAL_TESTS_REGISTER_REPORTER(Mac::VisualStudioCodeReporter);
APPROVAL_TESTS_REGISTER_REPORTER(Mac::CLionDiffReporter);
```

(See snippet source)

Edit tests/DocTest_Tests/reporters/CommandLineReporterTests.cpp

- Add an instance of the new Reporter class

```
// Mac
std::make_shared<Mac::AraxisMergeReporter>(),
std::make_shared<Mac::BeyondCompareReporter>(),
std::make_shared<Mac::DiffMergeReporter>(),
std::make_shared<Mac::KaleidoscopeReporter>(),
std::make_shared<Mac::P4MergeReporter>(),
std::make_shared<Mac::SublimeMergeReporter>(),
std::make_shared<Mac::KDiff3Reporter>(),
std::make_shared<Mac::TkDiffReporter>(),
std::make_shared<Mac::VisualStudioCodeReporter>(),
std::make_shared<Mac::CLionDiffReporter>(),
```

(See snippet source)

- Run this test, review and accept the changes.

6.7.2 Adding a new Windows reporter

The steps are the same as above, except that in the second step, you will edit:

- ApprovalTests/reporters/WindowsReporters.h
- ApprovalTests/reporters/WindowsReporters.cpp

6.7.3 Adding a new Linux reporter

The steps are the same as above, except that in the second step, you will edit:

- ApprovalTests/reporters/LinuxReporters.h
- ApprovalTests/reporters/LinuxReporters.cpp

6.7.4 Adding a new Cross Platform reporter

To add a reporter that will run on all 3 platforms, the steps are the same as above, except that in the second step, you will edit:

- ApprovalTests/reporters/CrossPlatformReporters.h
- ApprovalTests/reporters/CrossPlatformReporters.cpp

6.8 How to Toggle Enabling or Disabling of Deprecated Code

6.8.1 Introduction

> “To Err is Human, to Deprecate is Divine…”

Shakespeare (paraphrased)

From time to time, we realise that our previous decisions were a mistake, that we don’t want to carry with us in the future. We also don’t want to strand our users, who are currently relying on the old code working the way that it does.

Here’s how Approval Tests deals with deprecations.

6.8.2 Deprecation Mechanics

When enabled, these deprecation warnings will show up as:

- compiler C++14 and above, using the `[[deprecated("...")]]` feature
- messages on std::cout in C++11

6.8.3 Opting in

Show warnings

To opt in to warnings, add this line to your C++ code:

```
#define APPROVAL_TESTS_SHOW_DEPRECATION_WARNINGS 1
```

(See snippet source)

Or this to your CMakeLists.txt:

```
# Replace ${PROJECT_NAME} with the name of your test executable:
target_compile_definitions(${PROJECT_NAME} PRIVATE -DAPPROVAL_TESTS_SHOW_DEPRECATION_
↪WARNINGS=1)
```

(See snippet source)

Hide deprecated code

A more extreme version of this is to not even compile the deprecated code. You can do this by adding this line:

```
#define APPROVAL_TESTS_HIDE_DEPRECATED_CODE 1
```

(See snippet source)

Or this to your CMakeLists.txt:

```
# Replace ${PROJECT_NAME} with the name of your test executable:
target_compile_definitions(${PROJECT_NAME} PRIVATE -DAPPROVAL_TESTS_HIDE_DEPRECATED_
↪CODE=1)
```

(See snippet source)

6.8.4 How to Update Calls to Deprecated Code

Whenever we deprecate a method, the implementation of the deprecated method will always contain a single line, which is how we want the code to be called in the future.

As such, you can always open up the method to see how to convert your code.

If you IDE supports inlining, you can also select your old function call, and inline just that one line, and your IDE will update the code for you.

Note If you are reading this after we have removed the deprecated methods, please download a slightly earlier release, and then follow one of the steps above.

CHAPTER

SEVEN

BUILD SYSTEMS

- **Integrations**: *CMake* | *Conan*
- **Your builds**: *Build Machines and CI servers*

7.1 CMake Integration

7.1.1 Part 1: Reference

Integration Points

Because we use CMake to build ApprovalTests.cpp, we also provide integration points for our users.

CMake target

Approval Tests' CMake build exports an interface target `ApprovalTests::ApprovalTests`. Linking against it will add the proper include path and all necessary capabilities to the resulting binary.

This target is provided when ApprovalTests.cpp is used as a subdirectory. Assuming that ApprovalTests.cpp has been cloned to `lib/ApprovalTests.cpp`:

```
add_subdirectory(lib/ApprovalTests.cpp)
target_link_libraries(tests ApprovalTests::ApprovalTests)
```

CMake project options

ApprovalTests.cpp's CMake project also provides some options for other projects that consume it.

The file CMake/ApprovalTestsOptions.cmake defines these options.

The options provided are:

Which targets are built

Note that these are always enabled if this is the top-level project.

- `APPROVAL_TESTS_BUILD_TESTING`
  - When `ON`, the self-tests are run. Defaults to `OFF`.
- `APPROVAL_TESTS_BUILD_EXAMPLES`
  - When `ON`, the examples are run. Defaults to `OFF`.
- `APPROVAL_TESTS_BUILD_DOCS`
  - When `ON`, documentation files are generated, if Doxygen, Sphinx and mdsnippets are installed. Defaults to `OFF`.
  - These are not part of the `ALL` target.

Which third_party libraries are made available

Note that these are always included if this is the top-level project.

- `APPROVAL_TESTS_BUILD_THIRD_PARTY_CATCH2`
  - When `ON`, this project's copy of the Catch2 test framework is included. Defaults to `OFF`.
- `APPROVAL_TESTS_BUILD_THIRD_PARTY_DOCTEST`
  - When `ON`, this project's copy of the doctest test framework is included. Defaults to `OFF`.
- `APPROVAL_TESTS_BUILD_THIRD_PARTY_UT`
  - When `ON`, this project's copy of the Boost.UT test framework is included. Defaults to `OFF`.

Control the behaviour of our builds

- `APPROVAL_TESTS_ENABLE_CODE_COVERAGE`
  - When `ON`, Approval Test's own tests are run with code coverage enabled. This uses Lars Bilke's CodeCoverage.cmake. Defaults to `OFF`.

CMake commands support

- `add_subdirectory()`
  - See Use own ApprovalTests.cpp and Catch2 clones below, for an example using add_subdirectory().
  - **Use case:** This is typically for you have your own copy of the Approval Tests project directory that you want to re-use.
- `FetchContent`
  - See Make CMake clone ApprovalTests.cpp and Catch2 below, for an example using the FetchContent module.
  - The examples below use `FetchContent_MakeAvailable()`, which requires CMake 3.14 or above.
  - If you only have CMake 3.11 or above, see FetchContent (CMake 3.11+) for how to use `FetchContent_Populate()`.

- **Use case:** This is typically for when you want CMake to download a specific version of Approval Tests for you, behind the scenes.

- `ExternalProject`
  - With CMake before 3.11, see the ExternalProject module. The `FetchContent` examples below should help get started with `ExternalProject`.
  - **Use case:** This is typically for when you want CMake to download a specific version of Approval Tests for you, behind the scenes, and you are using an older version of CMake.
- `find_package()` - not supported
  - There is not yet support for find_package().

Single header or CMake Integration?

There are two main options for incorporating Approval Tests in to your project:

1. Download the single-header file from the Releases page and, typically, add the header to your source code.
2. Obtain a copy of the entire ApprovalTests.cpp repository and incorporate it in to your CMake build scripts.

Options for obtaining the repository typically include:

- cloning it
- forking it
- including it as a sub-repository
- having a build tool, such as CMake, download it for you automatically as part of your builds

CMake Integration Benefits

We recommend using the CMake integration route, which has several user benefits over the single header:

- Automatic prevention of most of the scenarios listed in Troubleshooting Misconfigured Build.
- Source-code compatibility with the single header download:
  - For convenience, we provide a wrapper header file ApprovalTests.hpp, which can be used to access all the features of this library, without having to know which features are provided by which header files.
- Flexibility in how many of the Approval Tests include files you include:
  - There is also the option to include just the headers in ApprovalTests/ that you use.
  - This may slightly improve build speeds.
- It may occasionally provide workarounds to bugs.

7.1.2 Part 2: Optional Explanatory Examples

Scenarios when using Approval Tests

Context

Example Project

Suppose you are writing some tests that use ApprovalTests.cpp with the Catch2 test framework.

Your top-level `CMakeLists.txt` file might look something like this:

```cmake
cmake_minimum_required(VERSION 3.14 FATAL_ERROR)
# This version is needed for FetchContent_Declare & FetchContent_MakeAvailable

project(fetch_content_approvaltests)

enable_testing()

add_subdirectory(dependencies)
add_subdirectory(tests)
```

(See snippet source)

The important thing to note, for following the examples below, is the `add_subdirectory(dependencies)` line. It makes CMake load a file `dependencies/CMakeLists.txt`.

Each example below shows a `dependencies/CMakeLists.txt`, for the corresponding scenario. All other code is identical between the example directories.

Here is this example project.

Example Tests

Your `tests/CMakeLists.txt` file might look something like this:

```cmake
add_executable(tests
        main.cpp
        tests.cpp
)
target_link_libraries(tests ApprovalTests::ApprovalTests Catch2::Catch2)

target_compile_features(tests PUBLIC cxx_std_11)
set_target_properties(tests PROPERTIES CXX_EXTENSIONS OFF)

add_test(
        NAME tests
        COMMAND tests)
```

(See snippet source)

This says that the libraries `ApprovalTests::ApprovalTests` and `Catch2::Catch2` are required by the `tests` executable.

Here is this example project's test directory.

Dependencies

How might you enable CMake to provide those libraries? In other words, what are the options for the contents of `dependencies/CMakeLists.txt`?

The next few sections describe some options.

Make CMake clone ApprovalTests.cpp and Catch2

Note: The files in this section can be viewed and downloaded from fetch_content_approvaltests_catch2.

The following is for when you just want ApprovalTests.cpp and Catch2 to be downloaded as part of your project's build. You don't particularly want to see their source code, although you're happy if your debugger steps in to them.

It also needs CMake 3.14 or above.

We use this `dependencies/CMakeLists.txt` file:

```
# Needs CMake 3.14 or above
include(FetchContent)

# -------------------------------------------------------------------
# ApprovalTests.cpp
FetchContent_Declare(ApprovalTests
        GIT_REPOSITORY https://github.com/approvals/ApprovalTests.cpp.git
        GIT_TAG master)

FetchContent_MakeAvailable(ApprovalTests)

# -------------------------------------------------------------------
# Catch2
FetchContent_Declare(Catch2
        GIT_REPOSITORY https://github.com/catchorg/Catch2.git
        GIT_TAG v2.11.1)

FetchContent_MakeAvailable(Catch2)
```

(See snippet source)

Note the `GIT_TAG` values: This tells CMake which revision of dependencies to use. The value can be a tag or a git commit ID. Here we use `master`, to always test our integrations with the latest Approval Tests code. However, it is generally recommended to pin your dependencies to specific versions, and test behaviour before updating to newer versions.

After CMake has generated the build files, the directory structure would look something like this, where the `cmake-build-debug` directory is the build space, and the `cmake-build-debug/_deps` contains the downloaded and built ApprovalTests.cpp and Catch2 repositories:

```
fetch_content_approvaltests_catch2/
  .git/
  cmake-build-debug/
    _deps/
      approvaltests-build/
      approvaltests-src/
      approvaltests-subbuild/
      catch2-build
      catch2-src
      catch2-subbuild
```

(continues on next page)

(continued from previous page)

```
  ...
CMakeLists.txt
dependencies/
  CMakeLists.txt
tests/
  ...
```

Make CMake clone ApprovalTests.cpp

Note: The files in this section can be viewed and downloaded from fetch_content_approvaltests.

The only difference between the previous example and this one is that here we use the Catch2 header that is in the ApprovalTests.cpp project.

We use this `dependencies/CMakeLists.txt` file:

```
# Needs CMake 3.14 or above
include(FetchContent)

# -------------------------------------------------------------------
# ApprovalTests.cpp
FetchContent_Declare(ApprovalTests
        GIT_REPOSITORY https://github.com/approvals/ApprovalTests.cpp.git
        GIT_TAG master)

# Tell the ApprovalTests CMake files that we want to use its copy of Catch2:
set(APPROVAL_TESTS_BUILD_THIRD_PARTY_CATCH2 ON CACHE BOOL "")

FetchContent_MakeAvailable(ApprovalTests)
```

(See snippet source)

We have set `APPROVAL_TESTS_BUILD_THIRD_PARTY_CATCH2` to `ON`, so that CMake will use the copy of Catch2 that is included in the ApprovalTests.cpp repository.

There are also options to enable use of ApprovalTests.cpp's copies of all other supported test frameworks except GoogleTest, including:

- `APPROVAL_TESTS_BUILD_THIRD_PARTY_DOCTEST`
- `APPROVAL_TESTS_BUILD_THIRD_PARTY_UT`

Use own ApprovalTests.cpp and Catch2 clones

Note: The files in this section can be viewed and downloaded from add_subdirectory_approvaltests_catch2.

Here, instead of getting CMake to download ApprovalTests.cpp and Catch2, we have got our own clones or forks of them, which we want to use with our own tests.

In this example, the directory structure looks like this:

```
ApprovalTests.cpp/
  .git/
  CMakeLists.txt
```

(continues on next page)

(continued from previous page)

```
  ...
Catch2/
  .git/
  CMakeLists.txt
  ...

add_subdirectory_approvaltests_catch2/
  .git/
  CMakeLists.txt
  dependencies/
    CMakeLists.txt
  tests/
```

We use this `dependencies/CMakeLists.txt` file:

```
# -------------------------------------------------------------------
# ApprovalTests.cpp
add_subdirectory(
        ../../ApprovalTests.cpp
        ${CMAKE_CURRENT_BINARY_DIR}/approvaltests.cpp_build
)

# -------------------------------------------------------------------
# Catch2
set(CATCH_BUILD_TESTING OFF CACHE BOOL "")
add_subdirectory(
        ../../Catch2
        ${CMAKE_CURRENT_BINARY_DIR}/catch2_build
)
```

(See snippet source)

Here we use `add_subdirectory()`. This works with older versions of CMake, unlike the `FetchContent` examples above.

The above was tested with CMake 3.8.

If your directory layout differed from the above, you would change the relative paths in the `add_subdirectory()` lines.

Using other supported test frameworks

To save space and repetition, the examples above only show the Catch2 test framework.

The same principles apply when using all the other test frameworks supported by ApprovalTests.cpp.

Scenarios when developing ApprovalTests.cpp

Developing ApprovalTests.cpp with test framework sources

Note: The files in this section can be viewed and downloaded from dev_approvals.

For Approval Tests project maintainers, it is useful to be able to edit and debug both this project and the test frameworks that it depends upon. It helps to be able to see the source of these frameworks, rather than just the single-header releases that are copied in to the third_party directory here.

This also allows us to checkout different commits of any of these projects.

Here we want to enable and run all the ApprovalTests.cpp tests, unlike the cases above, where we only want to run the tests of the project that is being developed.

Consider this directory structure, where the repositories for all these projects are checked out side-by-side, and there is an extra directory `dev_approvals/` that will contain just a `CMakeLists.txt` file, to set up a project containing all the other directories:

```
ApprovalTests.cpp/
  .git/
  CMakeLists.txt
  ...
Catch2/
  .git/
  CMakeLists.txt
  ...
doctest/
  .git/
  CMakeLists.txt
  ...
googletest/
  .git/
  CMakeLists.txt
  ...
ut/
  .git/
  CMakeLists.txt
  ...

dev_approvals/
  CMakeLists.txt
```

The file `dev_approvals/CMakeLists.txt` creates a kind of "super build": one project for developing Approval-Tests.cpp and all the projects it depends on:

```
cmake_minimum_required(VERSION 3.8 FATAL_ERROR)

project(dev_approvals)
```

(continues on next page)

(continued from previous page)

```
enable_testing()

set(CMAKE_VERBOSE_MAKEFILE off)

# Prevent ctest creating cluttering up CLion with nearly 30 CTest targets
# (Continuous, ContinuousBuild etc) when it does:
#   include(CTest)
# This hack taken from https://stackoverflow.com/a/57240389/104370
set_property(GLOBAL PROPERTY CTEST_TARGETS_ADDED 1) # hack to prevent CTest added targets

# -------------------------------------------------------------------
# boost
# This will be used by find_package() in ApprovalTests.cpp/tests/Boost_Tests
# If there is a local boost directory, use tat.
# Otherwise, require the user to have installed boost (as is done in CI builds)
if (EXISTS ${CMAKE_CURRENT_SOURCE_DIR}/../boost)
    set(BOOST_ROOT ${CMAKE_CURRENT_SOURCE_DIR}/../boost)
endif()

# -------------------------------------------------------------------
# Catch2
set(CATCH_BUILD_TESTING OFF CACHE BOOL "")
add_subdirectory(
        ../Catch2
        ${CMAKE_CURRENT_BINARY_DIR}/catch2_build
)

# -------------------------------------------------------------------
# CppUTest

# Prevent CppUTest's own tests from being built
set(TESTS OFF CACHE BOOL "")

# Prevent build of CppUTest from generating thousands of lines of
# -Wc++98-compat and -Wc++98-compat-pedantic warnings:
set(C++11 ON CACHE BOOL "Compile with C++11 support")

add_subdirectory(
        ../cpputest
        ${CMAKE_CURRENT_BINARY_DIR}/cpputest_build
)

# -------------------------------------------------------------------
# doctest
add_subdirectory(
        ../doctest
        ${CMAKE_CURRENT_BINARY_DIR}/doctest_build
)

# -------------------------------------------------------------------
# filesystem
```

(continues on next page)

(continued from previous page)

```
set(CATCH_BUILD_TESTING OFF CACHE BOOL "")
add_subdirectory(
        ../filesystem
        ${CMAKE_CURRENT_BINARY_DIR}/filesystem_build
)

# -------------------------------------------------------------------
# fmt
set(CATCH_BUILD_TESTING OFF CACHE BOOL "")
add_subdirectory(
        ../fmt
        ${CMAKE_CURRENT_BINARY_DIR}/fmt_build
)

# -------------------------------------------------------------------
# GoogleTest
# Prevent GoogleTest from overriding our compiler/linker options
# when building with Visual Studio
set(gtest_force_shared_crt ON CACHE BOOL "" )
add_subdirectory(
        ../googletest
        ${CMAKE_CURRENT_BINARY_DIR}/googletest_build
)

# -------------------------------------------------------------------
# Boost.ut
set(BUILD_BENCHMARKS OFF CACHE BOOL "")
set(BUILD_EXAMPLES OFF CACHE BOOL "")
set(BUILD_TESTS OFF CACHE BOOL "")

add_subdirectory(
        ../ut
        ${CMAKE_CURRENT_BINARY_DIR}/ut_build
)

if(TARGET Boost::ut)
    add_library(boost.ut ALIAS ut)
endif()

if ("${CMAKE_CXX_COMPILER_ID}" MATCHES "Clang")
    # Turn off some checks off for boost.ut
    target_compile_options(ut INTERFACE
            -Wno-c99-extensions # Needed for Boost.ut, at least in v1.1.6
            -Wno-documentation-unknown-command # unknown command tag name \userguide
            -Wno-weak-vtables
            -Wno-comma # See https://github.com/boost-ext/ut/issues/398
            )
endif()

# -------------------------------------------------------------------
# ApprovalTests.cpp
```

(continues on next page)

(continued from previous page)

```
set(APPROVAL_TESTS_BUILD_TESTING ON CACHE BOOL "")
set(APPROVAL_TESTS_BUILD_EXAMPLES ON CACHE BOOL "")

add_subdirectory(
        ../ApprovalTests.cpp
        ${CMAKE_CURRENT_BINARY_DIR}/approvaltests.cpp_build
)
```

(See snippet source)

7.2 Conan Integration

7.2.1 Using Conan to obtain ApprovalTests.cpp

The Conan C++ package manager knows how to download released versions of ApprovalTests.cpp, and integrate the downloaded single-header file in to various C++ build systems.

This page assumes basic familiarity with Conan. For more information, see Conan's extensive documentation.

7.2.2 Example Conan CMake Setups

These examples demonstrate a few different ways of using Conan with ApprovalTests.cpp. They differ in which Conan generator they use.

They all specify their dependencies in a conanfile.txt file, but they could just as easily use a conanfile.py instead.

Example 1. Using Conan's cmake_find_package and cmake_paths generators

> **Scenario:** I want to use CMake's `find_package()` and have Conan obtain the packages for me. I only want the Conan references to appear at the top of my project.

Note: The files in this section can be viewed and downloaded from conan_cmake_find_package.

This example use Conan's cmake_find_package generator, and optionally also the cmake_paths generator.

The benefit of these generators is consistency: the target names for dependencies (for example, `ApprovalTests::ApprovalTests`) are generally the same as you would get if building against the library's own CMake files.

This gives more flexibility, as it opens up the possibility of some users obtaining dependencies via Conan, and other users building the dependencies themselves, with CMake)

The `conanfile.txt` file lists the required libraries, and which generator to use (here, `cmake_find_package` and optionally `cmake_paths`):

```
[requires]
catch2/2.13.4
approvaltests.cpp/10.7.0

[generators]
cmake_find_package
cmake_paths
```

(See snippet source)

There are two choices for the CMake instructions used in the top-level CMakeLists.txt file with these generators, as explained in the comments here:

```
cmake_minimum_required(VERSION 3.14 FATAL_ERROR)

project(conan_cmake_find_package)

# EITHER Using the "cmake_find_package" generator
#set(CMAKE_MODULE_PATH ${CMAKE_BINARY_DIR} ${CMAKE_MODULE_PATH})
#set(CMAKE_PREFIX_PATH ${CMAKE_BINARY_DIR} ${CMAKE_PREFIX_PATH})

# OR Using the "cmake_find_package" and "cmake_paths" generators
include(${CMAKE_BINARY_DIR}/conan_paths.cmake)

find_package(Catch2 REQUIRED)
find_package(ApprovalTests REQUIRED)

enable_testing()

add_subdirectory(tests)
```

(See snippet source)

And the CMakeLists.txt that builds the tests is as follows (note the standard library target names):

```
add_executable(tests
        main.cpp
        tests.cpp
)
target_link_libraries(
        tests
        ApprovalTests::ApprovalTests
        Catch2::Catch2)

target_compile_features(tests PUBLIC cxx_std_11)
set_target_properties(tests PROPERTIES CXX_EXTENSIONS OFF)

add_test(
        NAME tests
        COMMAND tests)
```

(See snippet source)

Example set of build commands to download dependencies, make the test program and run the tests:

```
#!/bin/bash

# Force execution to halt if there are any errors in this script:
set -e
set -o pipefail

mkdir -p build
cd       build
```

(continues on next page)

(continued from previous page)

```
conan install ..
cmake -DCMAKE_BUILD_TYPE=Debug ..
cmake --build .
ctest --output-on-failure . -C Debug
```

(See snippet source)

Example 2. Using Conan's cmake generator

> **Scenario:** I'm only going to be building with Conan, so I don't mind Conan-specific libraries appearing in `target_link_libraries()` in CMake: I just want my top-level CMake files to be simple - not cluttered with find_packages().

Note: The files in this section can be viewed and downloaded from conan_cmake.

This example use Conan's cmake Generator.

The `conanfile.txt` file lists the required libraries, and which generator to use (here, `conan`):

```
[requires]
catch2/2.13.4
approvaltests.cpp/10.7.0

[generators]
cmake
```

(See snippet source)

The `conan` generator generates a `conanbuildinfo.cmake` file, which needs to used in the top-level CMakeLists.txt file like this:

```
cmake_minimum_required(VERSION 3.14 FATAL_ERROR)

project(conan_cmake)

# Conan's cmake generator creates a conanbuildinfo.cmake file, which we
# need to include, and then use:
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup(TARGETS)

enable_testing()

add_subdirectory(tests)
```

(See snippet source)

And the CMakeLists.txt that builds the tests is as follows (note the Conan-specific library target names):

```
add_executable(tests
        main.cpp
        tests.cpp
)

# Note the Conan-specific library namees, beginning with CONAN_PKG.
```

(continues on next page)

(continued from previous page)

```
# Conan sets up these names when its cmake generator is used.
# This ties your project to using Conan.
target_link_libraries(
        tests
        CONAN_PKG::approvaltests.cpp
        CONAN_PKG::catch2)

target_compile_features(tests PUBLIC cxx_std_11)
set_target_properties(tests PROPERTIES CXX_EXTENSIONS OFF)

add_test(
        NAME tests
        COMMAND tests)
```

(See snippet source)

Example set of build commands to download dependencies, make the test program and run the tests:

```
#!/bin/bash

# Force execution to halt if there are any errors in this script:
set -e
set -o pipefail

mkdir -p build
cd       build
conan install ..
cmake -DCMAKE_BUILD_TYPE=Debug ..
cmake --build .
ctest --output-on-failure . -C Debug
```

(See snippet source)

Example 3. Making CMake invoke Conan

> **Scenario:** I want to use CMake without having to remember to run a Conan command to make it download my dependencies.
>
> This will mean your dependencies are always uptodate (at the cost of a slightly slower build)

Note: The files in this section can be viewed and downloaded from cmake_invoking_conan.

This example use Conan's cmake-conan CMake module.

An advantage of this approach is that a project can use Conan to download dependencies, without people building that needing to know to run `conan install`. Anyone who is used to using CMake to generate builds will be able to build projects that use this mechanism. There will still need to be an installation of Conan on the build machine, however.

The `conanfile.txt` file lists the required libraries but does not say which generator to use:

```
# See CMake/Conan.cmake for how 'conan install' is launched from cmake

[requires]
catch2/2.13.4
approvaltests.cpp/10.7.0
```

(continues on next page)

(continued from previous page)

```

# Note that we don't say what generator we want.
# CMake code will take care of that for us.
```

(See snippet source)

There is a CMake file called `CMake/Conan.cmake` which contains instructions for downloading a specific version of the cmake-conan CMake module:

```
macro(run_conan)
# Download automatically, you can also just copy the conan.cmake file
if(NOT EXISTS "${CMAKE_BINARY_DIR}/conan.cmake")
  message(
    STATUS
      "Downloading conan.cmake from https://github.com/conan-io/cmake-conan")
  file(DOWNLOAD "https://github.com/conan-io/cmake-conan/raw/v0.15/conan.cmake"
       "${CMAKE_BINARY_DIR}/conan.cmake")
endif()

include(${CMAKE_BINARY_DIR}/conan.cmake)

conan_add_remote(NAME bincrafters URL
                 https://api.bintray.com/conan/bincrafters/public-conan)

conan_cmake_run(
  CONANFILE conanfile.txt
  BASIC_SETUP
  CMAKE_TARGETS # individual targets to link to
  BUILD
    missing
)
endmacro()
```

(See snippet source)

The top-level CMakeLists.txt file includes the above `CMake/Conan.cmake` file, and runs the macro that it contained:

```
cmake_minimum_required(VERSION 3.14 FATAL_ERROR)

project(conan_cmake)

# Load CMake/Conan.cmake, which sets up a 'run_conan()' macro to download dependencies.
include(CMake/Conan.cmake)
run_conan()

enable_testing()

add_subdirectory(tests)
```

(See snippet source)

And the CMakeLists.txt that builds the tests is as follows (note the Conan-specific library target names):

```
add_executable(tests
        main.cpp
        tests.cpp
)

# Note the Conan-specific library namees, beginning with CONAN_PKG.
# Conan sets up these names when its cmake generator is used.
# This ties your project to using Conan.
target_link_libraries(
        tests
        CONAN_PKG::approvaltests.cpp
        CONAN_PKG::catch2)

target_compile_features(tests PUBLIC cxx_std_11)
set_target_properties(tests PROPERTIES CXX_EXTENSIONS OFF)

add_test(
        NAME tests
        COMMAND tests)
```

(See snippet source)

Example set of build commands to download dependencies, make the test program and run the tests - note that there isno line to run conan:

```
#!/bin/bash

# Force execution to halt if there are any errors in this script:
set -e
set -o pipefail

mkdir -p build
cd       build
# Note that we do not need to invoke conan.
# However, we do need to say what build configuration we want.
cmake -DCMAKE_BUILD_TYPE=Debug ..
cmake --build .
ctest --output-on-failure . -C Debug
```

(See snippet source)

7.2.3 Other people's examples

Some examples of using the Conan package manager:

- Daniel Heater's ApprovalTests-ConanDemo repo
  - This demonstrates using ApprovalTests.cpp with Conan's `cmake` generator in a `conanfile.txt` file.
- p-podsiadly's ImageApprovals
  - This shows a different approach, using Conan's `cmake_find_package` generator from purely CMake code.
  - It's also a nice example of extending ApprovalTests.cpp to compare images.

7.2.4 Links

- For users: conan.io/center/approvaltests.cpp
- For maintainers: the Conan recipe: approvaltests.cpp

7.3 Vcpkg Integration

7.3.1 Using vcpkg to obtain ApprovalTests.cpp

Approval Tests is available via vcpkg, since v.10.9.1

Windows:

```
.\vcpkg install approval-tests-cpp
```

Linux/Mac:

```
vcpkg install approval-tests-cpp
```

7.3.2 Links

- For users: approval-tests-cpp in vcpkg.io
- For maintainers: the vcpkg port: approval-tests-cpp

7.4 Build Machines and Continuous Integration servers

In automated builds, if a file verification fails, there is no point opening up a graphical diff tool to show any errors. At best it is a waste of resources, and at worst, it may stop the build from completing.

So by default, Approval tests will never launch any graphical reporters on supported CI machines. To do this, we use Front Loaded Reporters.

Supported CI systems:

```
AppVeyor,
AzurePipelines,
GitHubActions,
GoCD
Jenkins,
TeamCity,
Travis,
```

(See snippet source)

The CI detection is based on environment variables, so it may also just work on other systems that we are unaware of:

```
"CI",
"CONTINUOUS_INTEGRATION",
"GITHUB_ACTIONS",
"GO_SERVER_URL",
```

(continues on next page)

(continued from previous page)

```
"JENKINS_URL",
"TEAMCITY_VERSION",
"TF_BUILD"
```

(See snippet source)

Or you may be able to set one of these environment variables in the configuration of your CI system, to tell Approval Tests it’s running under CI.

However, if your CI system is not supported, and you want to create a custom CI reporter, we suggest you start by looking at CIBuildOnlyReporter.

CHAPTER

EIGHT

EXTRAS

- **Various**: *Features* | *Why We Are Converting To Options* | *FAQs* | *Glossary* | *Utilities* | *Videos*

8.1 Features

8.1.1 v.x.y.z

8.1.2 v.10.12.0

EmptyFileCreatorByType::registerCreator

Empty file creation is now customizable for individual file extensions.

8.1.3 v.10.11.0

useFileNameSanitizer

You can now customize how invalid filename characters are converted. See Converting Test Names to Valid FileNames

useEmptyFileCreator

Empty file creation is now customizable, for when you are verifying non-text files.

8.1.4 v.10.10.0

Storyboard

Storyboard is a utility that allows you to print the changes to an object over time

Grid

Grid is a utility that creates 2D text in a grid format

Vcpkg Integration

See Vcpkg Integration.

8.1.5 v.10.9.0

Custom template namer

See TemplatedCustomNamer

Options.withNamer()

Options now has the ability to specify a custom namer.

Path

See Path.h and Path.cpp

8.1.6 v.10.8.0

Ninja builds work when inside source tree

The entire Misconfigured Builds page of workarounds is no needed, but is retained for those using older releases.

8.1.7 v.10.7.0

CombinationApprovals header

Optional header argument added to CombinationApprovals.

8.1.8 v.10.6.0

FrameworkIntegrations class

We now have a centralised portal, `FrameworkIntegrations`, to all the places that you need to use, in order to add support for a new test framework.

Approvals::verify() counts as an assertion in test frameworks

Previously, Approvals wouldn't register as an assertion, giving misleading messages and reporting, and sometimes causing a test framework to complain that there were no assertions.

8.1.9 v.10.5.0

Support for selecting Reporter at run-time

See How to Select a Reporter with an Environment Variable.

8.1.10 v.10.4.0

Support for CppUTest framework

See Using Approval Tests With CppUTest.

8.1.11 v.10.3.0

Approvals::verifyAll and std::initializer_list

Added `std::initializer_list` support to `Approvals::verifyAll()`.

For example:

```
ApprovalTests::Approvals::verifyAll({10, 20, 30});
```

(See snippet source)

8.1.12 v.10.2.0

DateUtils

See DateUtils

8.1.13 v.10.1.1

Improved Compilation Speeds

Fig. 1: Compilation Times: v.10.1.0: 2.7 secs vs v.10.1.1: 1.2 secs

8.1.14 v.10.0.0

Removed Deprecated Code

See Why We Are Converting To Options for easy ways to update your code.

Specifically, the following have been removed:

- Methods
  - `Approvals::verify(..., Reporter)`
  - `Approvals::verifyAll(..., Reporter)`
  - `Approvals::verifyExistingFile(..., Reporter)`
  - `Approvals::verifyExceptionMessage(..., Reporter)`
  - `Approvals::verifyWithExtension(...)`
  - `CombinationApprovals::verifyAllCombinations(Reporter, ...)`
- Macros
  - `APPROVAL_TESTS_HIDE_DEPRECATED_CODE`
  - `APPROVALTESTS_VERSION`
  - `APPROVALTESTS_VERSION_MAJOR`
  - `APPROVALTESTS_VERSION_MINOR`
  - `APPROVALTESTS_VERSION_PATCH`
  - `APPROVALTESTS_VERSION_STR`
  - `APPROVALS_CATCH_DISABLE_FILE_MACRO_CHECK`

8.1.15 v.8.9.0

Regex-based Scrubbing

See Scrubbing using Regular Expressions (regex).

8.1.16 v.8.8.0

FmtApprovals

See How to Use the Fmt Library To Print Objects.

8.1.17 v.8.7.0

Options

See Options.

Scrubbers

See How to Scrub Non-Deterministic Output.

StringMaker and TApprovals

This is in internal change, which will provide future flexibility, and does not change any existing code.

In this release, we:

- templatized the Approvals class, renaming it to TApprovals
- changed the CombinationApprovals namespace to a template class called TCombinationApprovals
- introduced the StringMaker class as an additional customization point for the above two classes

To learn about the extra string-formatting options for your objects, see To String.

Consistent macro names

All our Macros now start with `APPROVAL_TESTS_`.

We have kept the old macros, redirecting to the new ones, for backwards compatibility.

Old	New
`APPROVALTESTS_VERSION`	`APPROVAL_TESTS_VERSION`
`APPROVALTESTS_VERSION_MAJOR`	`APPROVAL_TESTS_VERSION_MAJOR`
`APPROVALTESTS_VERSION_MINOR`	`APPROVAL_TESTS_VERSION_MINOR`
`APPROVALTESTS_VERSION_PATCH`	`APPROVAL_TESTS_VERSION_PATCH`
`APPROVALTESTS_VERSION_STR`	`APPROVAL_TESTS_VERSION_STR`
`APPROVALS _CATCH_DISABLE_FILE_MACRO_CHECK`	`APPROVAL_ TESTS_DISABLE_FILE_MACRO_CHECK`

8.1.18 v.8.6.0

Support for Boost.Test framework

See Using Approval Tests With Boost.Test

8.1.19 v.8.5.0

Support for Sublime Merge

Added support for Sublime Merge on Linux, macOS, and Windows (See #103)

Support for Beyond Compare on Linux

Added support for Beyond Compare on Linux (See #114).

8.1.20 v.8.3.0

Flexibility for adding custom merge tools

`CustomReporter::create()` adds flexibility for adding custom merge tools: see How to Use A Custom Reporter.

Supporting new merge tools.

See How to Submit a New Reporter to ApprovalTests.

8.1.21 v.8.2.0

Conan Integration documented

See Conan Integration.

8.1.22 v.8.1.0

Version detection

ApprovalTests provides the following macros to detect the release version, with a set of example values shown in italics:

- `APPROVAL_TESTS_VERSION_MAJOR`: *8*
- `APPROVAL_TESTS_VERSION_MINOR`: *1*
- `APPROVAL_TESTS_VERSION_PATCH`: *2*
- `APPROVAL_TESTS_VERSION_STR`: *8.1.2*
- `APPROVAL_TESTS_VERSION`: *80102*

8.1.23 v.8.0.0

CMake Integration documented

See CMake Integration.

Use with Ninja generator documented

See Troubleshooting Misconfigured Build, for explanations, fixes and workarounds.

8.1.24 v.7.0.0

See the v.7.0.0 milestone for the full list of changes.

Support for [Boost].UT test framework

See Using Approval Tests With [Boost].UT.

8.1.25 v.6.0.0

Existing Catch Project - with your main()

See Existing Project - with your main().

8.1.26 v.5.1.0

Continuous Integration Builds

Approval tests will now never launch any reporters on supported Continuous Integration machines.

Approving multiple files in one test

See Approving multiple files in one test

ExceptionCollector

See ExceptionCollector

Using custom writers

See Using custom writers

Using custom filename extensions

See Using custom filename extensions

8.1.27 v.5.0.0

Multiple output files per test

See Multiple output files per test.

SeparateApprovedAndReceivedDirectoriesNamer

See SeparateApprovedAndReceivedDirectoriesNamer

Registering a Custom Namer

See Registering a Custom Namer

8.1.28 Before v.5.0.0

Customizing Google Tests Approval File Names

See Using Approval Tests With Google Tests

Blocking Reporter

See Blocking Reporters

Machine Blockers

Sometimes you will want tests to only run on certain machines. Machine blockers are a great way to do this.

```
TEST_CASE("Only run this test on John's machine")
{
    auto blocker = ApprovalTests::MachineBlocker::onMachinesNotNamed("JOHNS_MACHINE");
    if (blocker.isBlockingOnThisMachine())
    {
        return;
    }
    // Write tests here that depend on John's environment.
    REQUIRE(ApprovalTests::SystemUtils::getMachineName() == "JOHNS_MACHINE");
}
```

(See snippet source)

Front Loaded Reporters

See Front Loaded Reporters

Using sub-directories for approved files

See Using sub-directories for approved files

8.2 Why We Are Converting To Options

8.2.1 Introduction

We have introduced an optional `Options` parameter, instead of the optional `Reporter` parameter. The following is the history and reasoning for making this change, as well as our plans to roll it out.

For information on using Options itself, see Options.

8.2.2 Scrubbers

With the addition of Scrubbers, we realised that ApprovalTests has some optional parameters. The only one we had of these before was reporter. Because of that, we decided to group together all of the options in to a single container object. This gives us a few advantages:

1. We can expand in the future without changing the API.
2. Adding functionality becomes simpler, because it passes through to a single place.

It is effectively preparing to use the "Introduce Parameter Object" refactoring pattern.

8.2.3 API

Note: The following series of releases was completed with the release of v.10.0.0. We are retaining this documentation to help out anyone who has yet to update from one of the earlier release.

Our current pattern is to have an optional Reporter at the end of any verify() method.

We are switching this to have an option `Options` object instead.

This temporarily doubles our API interface, and we are deprecating the Reporter overloads. When enabled, these deprecation warnings will show up as:

- compiler C++14 and above, using the `[[deprecated("..."]]` feature
- messages on std::cout in C++11

8.2.4 The Plan

Historically, we have found that it is easier for users to update code for breaking changes if these changes are rolled out in a graduated way. This allows users to select which version of the library to use, to have the ability to update code incrementally.

We how now finished our short series of quick releases to release this:

1. deprecation warnings are off: users can opt-in (v.8.7.0)
2. deprecation warnings are on: users can opt-out (v.8.9.1)
3. deprecation warnings are forced, code still exists (v.8.9.2)
4. the deprecated methods are hidden: users can opt-in (v.9.0.0)
5. the deprecated methods are removed (v.10.0.0)

Step	Deprecation Warnings	Deprecated Code
	See `APPROVAL_ TESTS_SHOW_DEPRECATION_WARNINGS`	See `APPROVAL _TESTS_HIDE_DEPRECATED_CODE`
1	Optional: off	Optional: enabled
2	Optional: on	Optional: enabled
3	Always on	Optional: enabled
4	Always on	Optional: hidden
5	Not applicable	Removed

Suggested strategy

We suggest that any time you want to remove the deprecations, you jump ahead and toggle hide-deprecations, and let the compiler help you find the code you need to update.

8.2.5 Opting in

Currently (2020-06-04), all deprecated code is removed.

If you are are using an earlier version, see How to Toggle Enabling or Disabling of Deprecated Code.

8.2.6 How to Update Calls to Deprecated Code

Whenever we deprecate a method, the implementation of the deprecated method will always contain a single line, which is how we want the code to be called in the future.

As such, you can always open up the method to see how to convert your code.

If you IDE supports inlining, you can also select your old function call, and inline just that one line, and your IDE will update the code for you.

Note If you are reading this after we have removed the deprecated methods, please download a slightly earlier release, and then follow one of the steps above.

Updating verify(..., Reporter)

Instead of passing in a `Reporter` instance, you are now going to pass in an `Options` object containing the Reporter instance, for example `Options(MyReporter())`.

This is an example what the new code would look like:

```
using namespace ApprovalTests;
Approvals::verify("text to be verified", Options(Windows::AraxisMergeReporter()));
```

(See snippet source)

Updating verifyWithExtension(..., fileExtensionWithDot, Reporter)

Before, we had several overloads of a special method - `verifyWithExtension()` - to set the file extension to be used when verifying a piece of text, for when `.txt` was not appropriate.

Because `Options` allows the file extension to be specified, all verify methods now have this capability.

As such, this specialised method is redundant, and is being removed.

This is an example what the new code would look like:

```
ApprovalTests::Approvals::verify(
    "<h1>hello world</h1>",
    ApprovalTests::Options().fileOptions().withFileExtension(".html"));
```

(See snippet source)

8.3 Frequently Asked Questions

If you would like us to add any more questions here, please contact us via the Contributing page.

8.3.1 Overview

What are approval tests?

How do they differ from unit tests?

Why are approval tests particularly good for testing legacy code?

8.3.2 Test Frameworks

Do I need to use the Catch2 in the Approval Tests repo?

No. The intention is that you can provide your own copy of Catch2, via a file call `catch.hpp`.

We use the copy of Catch2 and other test frameworks in third_party/ in this project only to run our own tests.

Certainly, if you download a Single Header release of this library, no Catch2 is provided, so you need to provide your own.

Does it integrate with other unit testing libraries?

8.3.3 Writing Tests

I wrote a test, but the output file has loads of stuff I'm not interested in

Things to say:

- Yes, it's a common problem
- Readability of the output is important
- Someone reviewing a test failure needs to understand the purpose and intent of the test
- **Recommendation**: write your own formatting that's specific to particular tests - see
  - How to do String Conversions
  - Tips for Designing Strings for examples.

I want to test images

You may find that your tests fail, even though equivalent Approved and Received files are being compared, if the image file formats being used encode things like the date the file was created. This is because ApprovalTests.cpp's default behaviour is a character-for-character comparison of file content.

If you can use the Qt framework, then we have provided a way to verify the contents of PNG images: please see `ApprovalTestsQt::verifyQImage()` in ApprovalTests.cpp.Qt.

8.4 Glossary

8.4.1 Approving Results

8.4.2 Approved File

8.4.3 Chain of responsibility (pattern)

8.4.4 Code Coverage

8.4.5 Combination Testing

Sometimes referred to as Combinatorial testing.

See Testing Combinations.

8.4.6 Comparator

See Custom Comparators.

8.4.7 Continuous Integration

8.4.8 Convention over Configuration

Wikipedia Entry

Instead of asking the user to specify everything, we make assumptions based on common patterns, so code usually just works "out of the box". This tends to dramatically reduce the amount of clutter, makes things easier, and reduces the amount of surprises.

8.4.9 Custom Asserts

8.4.10 Diff Tool

8.4.11 Disposable Objects

Objects that implement the *RAII* pattern.

See Disposable Objects.

8.4.12 Edge Case

8.4.13 Fake It Till You Make It

A development technique for building in small steps

8.4.14 Happy Path

8.4.15 Kata

8.4.16 Koans

8.4.17 Mutation Testing

8.4.18 Namer

8.4.19 Principle of Least Surprise

8.4.20 RAII (Resource acquisition is initialization)

This is a pattern where your object constructor opens a resource, such as memory, and your object destructor closes the resource.

This is also known as "Scope based resource management".

Wikipedia Entry

8.4.21 Received File

8.4.22 Reporter

See Reporters.

See Using sub-directories for approved files

See Features - whose sections need to be moved around

8.4.23 Scrubber

See Scrubbers.

8.4.24 Stringification

See String conversions.

8.4.25 System Under Test

The area of the production code that you are testing. See System Under Test.

8.4.26 Test Framework

8.4.27 test && commit || revert (TCR)

8.4.28 Writer

8.4.29 Yak Shaving

8.4.30 Sayings

- The tests test the code, and the code tests the tests
- Test until bored

8.5 Utilities

8.5.1 What is this?

There are many random small functions and classes that we create and use while writing Approvaltests.
Some of these make writing **code** easier, some of them make writing **tests** easier.

All of them are shared here, in no particular order, in the hope you might also benifit from them.

8.5.2 List of Utilities

- DateUtils: Create and print C++11 dates and times.
- ExceptionCollector: Collect multiple exceptions and throw a combined exception.
- Grid: Create 2D text in a grid format
- Storyboard: Print the changes to an object over time

8.6 Videos

8.6.1 Quickly Testing Legacy Code - by Clare Macrae

C++ on Sea, February 2019

The video Quickly Testing Legacy Code gives an introduction both to Approval Tests and to this library. *Given at C++ on Sea in February 2019*.

Slides: in PowerPoint and PDF formats.

Sample code: on Github.

CPPP, June 2019

The video Quickly Testing Legacy Code gives an introduction both to Approval Tests and to this library, covering some features that were added since the previous version. *Given at CPPP in Paris in June 2019*.

Slides: in PPT and PDF formats.

Sample code: on GitHub.

8.6.2 Quickly Testing Legacy C++ Code with Approval Tests - by Clare Macrae

CppCon, September 2019

The video Quickly Testing Legacy C++ Code with Approval Tests gives a newer introduction both to Approval Tests and to this library, covering many features implemented since the earlier talks. *Given at CppCon in September 2019*.

Slides: in PPT and PDF formats.

Sample code: on GitHub.

CHAPTER

NINE

TROUBLESHOOTING

- **Topics**: *Troubleshooting* | *Misconfigured Builds* | *Misconfigured main()*

9.1 Troubleshooting

9.1.1 Test gives "You have forgotten to configure your test framework. . . "

Symptom

Running tests gives the following output:

```
***************************************************************************
*                                                                         *
* Welcome to Approval Tests.
*
* You have forgotten to configure your test framework for Approval Tests.
*
* To do this in Catch, add the following to your main.cpp:
*
*     #define APPROVALS_CATCH
*     #include "ApprovalTests.hpp"
*
* To do this in Google Test, add the following to your main.cpp:
*
*     #define APPROVALS_GOOGLETEST
*     #include "ApprovalTests.hpp"
*
* To do this in doctest, add the following to your main.cpp:
*
*     #define APPROVALS_DOCTEST
*     #include "ApprovalTests.hpp"
*
* To do this in Boost.Test, add the following to your main.cpp:
*
*     #define APPROVALS_BOOSTTEST
*     #include "ApprovalTests.hpp"
*
* To do this in CppUTest, add the following to your main.cpp:
*
*     #define APPROVALS_CPPUTEST
```

(continues on next page)

(continued from previous page)

```
*     #include "ApprovalTests.hpp"
*
* To do this in [Boost].UT, add the following to your main.cpp:
*
*     #define APPROVALS_UT
*     #include "ApprovalTests.hpp"
*
* For more information, please visit:
* https://github.com/approvals/ApprovalTests.cpp/blob/master/doc/
↪TroubleshootingMisconfiguredMain.md
*                                                                             *
*******************************************************************************
```

(See snippet source)

Things to check:

See Troubleshooting Misconfigured Main.

9.1.2 Test gives "There seems to be a problem with your build configuration"

Symptom 1: Compilation Error

Compiling tests in Ninja-generated builds gives a compilation failure, with this message:

```
"There seems to be a problem with your build configuration, probably with Ninja. "
"Please visit https://github.com/approvals/ApprovalTests.cpp/blob/master/doc/
↪TroubleshootingMisconfiguredBuild.md "
"The filename is: "
__FILE__
```

Symptom 2: Test Failure

Running tests in Ninja-generated builds gives output such as the following:

```
*******************************************************************************
*                                                                             *
* Welcome to Approval Tests.
*
* There seems to be a problem with your build configuration.
* We cannot find the test source file at:
*   ../../../tests/Catch2_Tests/ApprovalsTests.cpp
*
* For details on how to fix this, please visit:
* https://github.com/approvals/ApprovalTests.cpp/blob/master/doc/
↪TroubleshootingMisconfiguredBuild.md
*
* For advanced users only:
* If you believe you have reached this message in error, you can bypass
* the check by calling ApprovalTestNamer::setCheckBuildConfig(false);
*                                                                             *
*******************************************************************************
```

(See snippet source)

Things to check:

See Troubleshooting Misconfigured Build.

9.1.3 My custom reporter works in development, but not CI

Check your test code - especially your main - for any uses of `Approvals::useAsFrontLoadedReporter()` that are specific to running on a Continuous Integration system.

If that's the case, and you do still want to use a custom reporter in an individual test, you can use `Approvals::useAsFrontLoadedReporter()` in the test, passing in your custom reporter, to take precedence over the CI-specific reporter in your main.

9.1.4 Running Catch2 tests in CLion gives 'unexpected exception'

Note: Updating to CLion 2020.2 EAP or later fixes this bug.

If there are problems with code that uses Approval Tests, or test failures, the library takes care to issue helpful information via the text in an exception (via `exception.what()`). This text is then displayed by the test framework.

However, some Catch2 users have reported not always seeing these messages, and instead seeing output like:

```
.../TestFileName.cpp:32: Failure:
unexpected exception
```

This turns out to be due to a bug in CLion's Catch2 plugin: Catch2 unexpected exception instead of detailed message.

There are various options to work around this:

- Try updating CLion or reviewing the linked ticket above, to see if the CLion problem has been fixed.
- Run the test executable in a console window, instead of in CLion, to see what the error is, and fix it.
- Use CLion's "Edit Configurations" to run the test program as a "CMake Application" instead of via the "Catch" test runner.

9.2 Troubleshooting Misconfigured Build

9.2.1 Version v.10.8.0

Significant improvements regarding the Ninja build system were made in Approval Tests v.10.8.0. Our suggestion is that if you are using an older version, start by upgrading.

9.2.2 Before v.10.8.0

Feedback Requested

This is living documentation. If you discover extra scenarios or better solutions, please contribute back via bug reports or pull requests. Thank you.

Symptoms

Compilation Error

Prior to v.10.8.0, compiling tests in Ninja-generated builds gives a compilation failure, with this message:

```
"There seems to be a problem with your build configuration, probably with Ninja. "
"Please visit https://github.com/approvals/ApprovalTests.cpp/blob/master/doc/
↪TroubleshootingMisconfiguredBuild.md "
"The filename is: "
__FILE__
```

Test Failure

Prior to v.10.8.0, running tests gives output such as the following:

```
*****************************************************************************
*                                                                           *
* Welcome to Approval Tests.
*
* There seems to be a problem with your build configuration.
* We cannot find the test source file at:
*   ../../../tests/Catch2_Tests/ApprovalsTests.cpp
*
* For details on how to fix this, please visit:
* https://github.com/approvals/ApprovalTests.cpp/blob/master/doc/
↪TroubleshootingMisconfiguredBuild.md
*
* For advanced users only:
* If you believe you have reached this message in error, you can bypass
* the check by calling ApprovalTestNamer::setCheckBuildConfig(false);
*                                                                           *
*****************************************************************************
```

(See snippet source)

The problem

Ninja generator

Approval Tests depends on the test framework to provide access to the full path of the source file of the test being run.

In many cases, this is implemented using `__FILE__`.

With some build configurations, we have found that the path contained in `__FILE__` contains either just the file name, or contains an incorrect relative path to a non-existent directory, relative to the current working directory of the test program.

This is what we have established:

- Some compilers only put a relative path in to `__FILE__`, if the filename they are given on the command line was relative
- On all platforms, the Ninja build generator:
  - gives the compiler **relative paths**, if the build tree is inside the source tree
  - gives the compiler **absolute paths** if build tree is outside the source tree
- Visual Studio recently changed its default generator to Ninja, making the problem much more common.

This means that **if Ninja is used to create a build-space that is inside the source tree, Approval Tests-based tests will fail**.

Note that Visual C++ has a way to over-ride this and force absolute paths, if given `/FC`. This is described below, in Situation: Visual Studio with Visual C++ compiler (cl.exe).

Solutions

This section lists the known types of workaround for the above problems.

Use a Ninja Unity build, if you can

When used to generate Unity builds, the Ninja build generator creates executables that run correctly with Approval Tests, finding the source file location correctly.

For more information about Unity builds in CMake, see this article: CMake 3.16 added support for precompiled headers & unity builds - what you need to know. It also describes the initial problems you may encounter when trying to compile a project as Unity, and how to fix them.

If you are using **CMake 3.16 or above**, it is easy to turn on Unity builds, and the following examples show you how.

Using the ApprovalTests.cpp repo

Here is an example CMake command-line for creating Unity builds with the Ninja generator:

```
# CMake 3.16 or above:
cmake -G "Ninja" -DCMAKE_UNITY_BUILD=yes <source_location>
```

In the following situations, the above is all you need to do, for all the supported test frameworks to work correctly:

- You are building the ApprovalTests.cpp project with CMake.
- You have added the ApprovalTests.cpp project to your own CMake build, via `add_subdirectory()`. For more information on the options for this, see CMake Integration.

Using the ApprovalTests.cpp single-header download

If you are using the single-header download of Approval Tests with a Ninja Unity build, you may find that you get a compilation failure, pointing to this page, when in fact the tests would run correctly.

In this case, you will need to disable the compilation check of `__FILE__`, which can be done by defining the `APPROVAL_TESTS_DISABLE_FILE_MACRO_CHECK` macro. This can be done line this:

```
# CMake 3.16 or above - with other frameworks, including Catch2, doctest, Google Test:
cmake -G "Ninja" -DCMAKE_UNITY_BUILD=yes \
    -DCMAKE_CXX_FLAGS_INIT=-DAPPROVAL_TESTS_DISABLE_FILE_MACRO_CHECK \
    <source_location>
```

Move your build outside the source tree

The problem with Ninja builds generating relative paths to source files only occurs if the build is inside the source tree.

If you are able to move your build outside the source tree, Ninja will generate absolute paths to source files, and Approval Tests will work fine.

If you need help to do this, see the various sections in Specific Situations below.

Force the compiler to use full-paths in `__FILE__`

This can be done with Visual C++: see below.

With [Boost].UT, check compiler and build options

The [Boost].UT framework uses very recent features of C++, and is changing somewhat rapidly.

If the ApprovalTests.cpp integration with [Boost].UT is not working in your build, you will see probably the following output at run-time, where the filename is `unknown`:

```
*****************************************************************************
*                                                                           *
* Welcome to Approval Tests.
*
* There seems to be a problem with your build configuration.
* We cannot find the test source file at:
*   unknown
*
* For details on how to fix this, please visit:
* https://github.com/approvals/ApprovalTests.cpp/blob/master/doc/
↪TroubleshootingMisconfiguredBuild.md
*
* For advanced users only:
* If you believe you have reached this message in error, you can bypass
* the check by calling ApprovalTestNamer::setCheckBuildConfig(false);
*                                                                           *
*****************************************************************************
```

(See snippet source)

These are possible causes:

- Check that your compiler and build options satisfy the requirements for using Approval Tests With [Boost].UT.
- If you have downloaded your own copy of the [Boost].UT framework, it's possible that it is not compatible with the ApprovalTests.cpp version you are using. It will be worth comparing your version of the [Boost].UT header with the one in this project: third_party/ut/include/boost/ut.hpp.

Use a non-Ninja generator

If you can't use any of the above, your only option to work around this Ninja limitation is probably to switch to a different CMake generator than Ninja.

Specific Situations

Situation: Visual Studio with Visual C++ compiler (cl.exe)

Use `/FC` to make Visual Studio emit the full path in diagnostics, and `__FILE__` (documentation).

If you are using this project by cloning or forking its repository, and then using CMake's `add_subdirectory()`, this will be done for you automatically, for all targets that use the provided CMake target.

Otherwise (such as if you are using the downloaded single-header file, or you have set up your own CMake builds), you need to add a line like the following to your `CMakeLists.txt` file:

```
if(CMAKE_CXX_COMPILER_ID STREQUAL "MSVC")
    target_compile_options(my_program_name PUBLIC /FC)
endif()
```

Or this:

```
target_compile_options(my_program_name PUBLIC $<$<CXX_COMPILER_ID:MSVC>:/FC>)
```

Situation: Visual Studio with Clang compiler (clang-cl.exe)

We have not been able to find a compiler flag that makes clang-cl put full paths in `__FILE__`.

The only solution we have found is to put your build outputs in a directory outside the source tree, so that the build will use absolute paths.

One way to do this is to edit your `CMakeSettings.json` file, and change all pairs of lines like this:

```
"buildRoot": "${projectDir}\\out\\build\\${name}",
"installRoot": "${projectDir}\\out\\install\\${name}",
```

To something like this (where you change `MyProjectName` to the actual name of your project):

```
"buildRoot": "${env.USERPROFILE}\\CMakeBuilds\\MyProjectName\\build\\${name}",
"installRoot": "${env.USERPROFILE}\\CMakeBuilds\\MyProjectName\\install\\${name}",
```

This would put the build outputs in to:

`C:\Users\YourUserName\CMakeBuilds\MyProjectName\build`

Situation: CMake's Ninja Generator

The easiest solution is probably to use a different CMake generator instead of Ninja.

However, if you wish to continue using Ninja, you will need to create and use a build directory outside of your source directory.

For example, with CMake, you might do this:

```
cd ApprovalTests.cpp
mkdir ../build_approval_tests_ninja
cd    ../build_approval_tests_ninja
cmake -G Ninja ../ApprovalTests.cpp
cmake --build .
ctest
```

9.3 Troubleshooting Misconfigured Main

9.3.1 Symptoms

Linker errors about missing symbols

When you link your test program - using ApprovalTest v.10.1.1 or later - you get linker errors about missing symbols, such as these:

- `ApprovalTests::FileApprover::verify(...)`
- `ApprovalTests::StringWriter::StringWriter(...)`
- `ApprovalTests::DefaultNamerFactory::getDefaultNamer()`
- `ApprovalTests::Options::defaultReporter()`
- `ApprovalTests::Scrubbers::doNothing(...)`
- `ApprovalTests::Options::FileOptions::getFileExtension() const`
- `ApprovalTests::Options::fileOptions() const`
- `ApprovalTests::Options::getReporter() const`
- `ApprovalTests::Options::scrub(...) const`

Error when running tests

Running tests - using ApprovalTest v.10.1.0 or earlier - gives the following output:

```
*****************************************************************************
*                                                                           *
* Welcome to Approval Tests.
*
* You have forgotten to configure your test framework for Approval Tests.
*
* To do this in Catch, add the following to your main.cpp:
*
```

(continues on next page)

(continued from previous page)

```
*     #define APPROVALS_CATCH
*     #include "ApprovalTests.hpp"
*
* To do this in Google Test, add the following to your main.cpp:
*
*     #define APPROVALS_GOOGLETEST
*     #include "ApprovalTests.hpp"
*
* To do this in doctest, add the following to your main.cpp:
*
*     #define APPROVALS_DOCTEST
*     #include "ApprovalTests.hpp"
*
* To do this in Boost.Test, add the following to your main.cpp:
*
*     #define APPROVALS_BOOSTTEST
*     #include "ApprovalTests.hpp"
*
* To do this in CppUTest, add the following to your main.cpp:
*
*     #define APPROVALS_CPPUTEST
*     #include "ApprovalTests.hpp"
*
* To do this in [Boost].UT, add the following to your main.cpp:
*
*     #define APPROVALS_UT
*     #include "ApprovalTests.hpp"
*
* For more information, please visit:
* https://github.com/approvals/ApprovalTests.cpp/blob/master/doc/
↪TroubleshootingMisconfiguredMain.md
*                                                                            *
******************************************************************************
```

(See snippet source)

9.3.2 Solutions

These are the things to check.

Check the instructions for your test framework

Approval Tests needs to know which test framework to connect to, and that usually involves making small changes to the `main()` function for your test program.

This error usually indicates that a problem in your test program's `main()` means that ApprovalTests.cpp is not correctly set up for your test framework.

The following resources should help:

- The text in the error message above may be enough to get you going.

- If not, see Getting Started - Creating your main() to find out what you need to do for your chosen test framework, or to select one, if you have not yet done so.

Check Google Test Framework

- Have you created a `main.cpp` that sets up ApprovalTests?
  - If not, the default Google Test `main()` will be used, which will not set up Approval Tests
  - To fix, copy in the non-comment code from tests/GoogleTest_Tests/main.cpp
- Is your `main.cpp` included in your project's build?
  - If not, the default Google Test `main()` will be used, which will not set up Approval Tests
  - To fix, e.g. check your `CMakeLists.txt` file
- Does your Google Test have its own custom `main.cpp`?
  - If so, perhaps you haven't yet added the code to set up Approval Tests?
  - To fix, copy in the Approvals-specific lines from examples/googletest_existing_main/main.cpp

Other Issues

- Is your code calling `Approvals::verify()` or any other methods in this library from outside a Google Test?
  - This is much less likely to be the cause, but the file-naming code in Approval Tests (`ApprovalTestNamer`) does require that approvals are used from inside a test method in a supported test framework.

CHAPTER

TEN

DEVELOPING APPROVALTESTS.CPP

- **Topics**: *Contributing to ApprovalTests.cpp* | *Coding Patterns* | *Maintaining the Docs* | *Building the Docs*

10.1 Contributing to ApprovalTests.cpp

10.1.1 Contributing - pairing and pull requests

The default way to add to most repositories is to fork and then create a pull-request.

This is **NOT** the default way to contribute to Approval Tests!

If you have something that you would like Approval Tests to have, including a bug you would like removed, we suggest you set up a pairing session (usually Skype or Facetime) with either Llewellyn or Clare or both.

A typical session lasts between 60 to 90 minutes. We do the work directly on master, on our own machines, with screen-sharing, and commit and push the results throughout the session. This results in quicker, higher-quality work. It gives us a better understanding of how people are using Approval Tests and the problems they encounter with it, and these sessions are also quite fun.

We use github's "co-author" feature, so everyone in the pairing session gets credit for the work. Virtually this entire project has been developed in this way, and is stronger for it.

Example co-author text, for use if not using the GitHub Desktop application:

```
Co-Authored-By: Llewellyn Falco <isidore@users.noreply.github.com>
```

We use Arlo's Commit Notation to prefix most commits, to indicate their level of risk.

Definition of Done

Note: Reminder for Clare and Llewellyn

- Tests
- Documentation
  - Sample code
  - Copy+pastable template (if appropriate)
  - A link on Features
  - A link on build/relnotes_x.y.z.md
  - Links on other appropriate places

- Check the Dashboard (after pushing)
- Retrospective (as an experiment) on what we learned from the work

10.1.2 Code of Conduct

Please note that this project is released with a Contributor Code of Conduct. By participating in this project you agree to abide by its terms.

10.1.3 Formatting Code

If possible, please configure your editor to use this repository's .clang-format file.

Instructions for this are available:

- CLion:
  - Settings/Preferences | Editor | Code Style | Turn on "Enable ClangFormat with clangd server" checkbox.
  - More information: ClangFormat as Alternative Formatter

There is a script to apply clang-format to the whole project: scripts/reformat_code.sh.

We have a CI job called `clang-format` that checks for code not correctly formatted: .github/workflows/github_actions_build.yml.

10.1.4 Updating the simulated single-header

After adding new header files to the library, we need to update ApprovalTests/ApprovalTests.hpp.

This is done by running scripts/create_simulated_single_header.sh:

```
cd ApprovalTests.cpp
./scripts/create_simulated_single_header.sh
```

10.1.5 Coding Patterns

See Static variables for header-only releases.

10.1.6 Documentation

We welcome improvements to the documentation! The page Maintaining the Docs describes how we manage the documentation files.

10.1.7 Releases

- Everything for releases is in the build directory
- There's more information in How to Release

10.1.8 Running shell scripts in cygwin

To run `.sh` in cygwin on Windows, add these lines to `~/.bash_profile`, and then re-start cygwin:

```
export SHELLOPTS
set -o igncr
```

Credit

10.2 Coding Patterns

10.2.1 Static variables for header-only releases

Note: As of v.10.1.1, this pattern is still used, but it is no longer needed. Method implementations are now in .cpp files, so the more conventional mechanism for initialising static variables would work fine.

Here is a sample of the pattern that we are using:

```
private:
    static std::shared_ptr<Reporter>& defaultReporter();

public:
    static std::shared_ptr<Reporter> getDefaultReporter();

    static void setDefaultReporter(const std::shared_ptr<Reporter>& reporter);
```

(See snippet source)

```
namespace ApprovalTests
{
    std::shared_ptr<Reporter>& DefaultReporterFactory::defaultReporter()
    {
        static std::shared_ptr<Reporter> reporter = std::make_shared<DiffReporter>();
        return reporter;
    }

    std::shared_ptr<Reporter> DefaultReporterFactory::getDefaultReporter()
    {
        return defaultReporter();
    }

    void
    DefaultReporterFactory::setDefaultReporter(const std::shared_ptr<Reporter>& reporter)
    {
```

(continues on next page)

(continued from previous page)

```
        defaultReporter() = reporter;
    }
}
```

(See snippet source)

Note the use of the reference (&) on the return type of the private method, and the addition of a getter and setter method.

10.3 Maintaining the Docs

10.3.1 Purpose of this page

We welcome improvements to the documentation! Here's how we manage the *content* of the documentation files.

For information on how that content is built and published, see Building the Docs.

10.3.2 Future version numbers

To refer to the next release, use 'v.x.y.z', and make sure that there is a step to edit the file to update that text to the actual version number in the release scripts.

10.3.3 Creating new pages

Using the template page manually

If creating a new Markdown page, please make a copy of doc/TemplatePage.md. This contains some boilerplate text which is tedious to create by hand.

Creating one or more pages by script

If creating multiple files, on Unix, you can use the script `doc/create_page.sh`

```
cd doc/
./create_page.sh TestingSingleObjects TestingContainers TestingCombinations
```

This won't overwrite existing files. It will write out the text to paste in to other .md files, to correctly link to the new file.

For documentation files outside of doc

If the new page will be outside of the doc folder, delete the following lines at the end:

```
---

[Back to User Guide](https://github.com/approvals/ApprovalTests.cpp/blob/master/doc/
↪README.md#top)
```

10.3.4 Linking to new pages

Two files need to be edited when new pages are added, so the new file is visible to users.

In Sphinx for Read the Docs

Each new documentation page needs to be added twice in doc/sphinx/index.rst.

1. A line beginning `:doc:`, which includes the page in the Read the Docs front page.
2. A link in a `.. toctree::` section, which includes the page in the navigation panel.

Note that Sphinx detects a page's title automatically, and will use it for the text in the hyperlink automatically, so you only need to spell out the link's text if you want it to be different from the page's title.

In the GitHub User Guide

Each new documentation page needs to be added to doc/README.md.

The layout here should mimic the layout generated you used in `index.rst`.

10.3.5 Internal links need to be absolute

All references to other files in this project, such as hyperlinks and images, must specify the full path from the root of the repository. This is needed for links to work correctly on the Read the Docs.

For example, use this:

```
* [this link will work everywhere](/doc/Reporters.md#top)
```

Not this:

```
* [this link is wrong](doc/Reporters.md#top)
```

And not this:

```
* [this link is wrong](Reporters.md#top)
```

10.3.6 Adding code and file samples

We use Simon Cropp's MarkdownSnippets tool to embed source code and other files in Markdown pages.

How it works

- See the MarkdownSnippets documentation for how to:
  - annotate snippets of source code,
  - reference the snippets in documentation.
- Run `run_markdown_templates.sh` **before commit**, every time a `.md` file or any of the source code with snippets is updated
  - See run_markdown_templates.sh.

- This will update the tables of contents and and any snippets in all .md files in the project.
- If this does not work, see that script for how to install the tools it uses

Managing CMake code samples

The repository claremacrae/ApprovalTests.cpp.CMakeSamples has some sample CMake projects for using ApprovalTests.cpp in various development scenarios.

When I improve the files in that repo, I run its script claremacrae/ApprovalTests.cpp.CMakeSamples/create_markdown.py to convert the interesting CMake files to Markdown.

Some of those Markdown files are then embedded in the documentation for this project.

- See CMakeIntegration.md for the final result.

10.3.7 Checking the documentation

Automated checking of links in documentation

There is a "markdown-link-check" github workflow that checks for broken links all in the Markdown files. It checks for missing file names, but will not detect missing anchors.

- It runs the script scripts/check_links.sh ...
- ... which uses the configuration file mlc_config.json ...
- ... and runs tcort's markdown-link-check

If there are any failures, the output is slightly verbose to look through - you have to find lines beginning `[*]`, but I think that these will be sufficiently few and far between that this is good enough.

Other checks of documentation

The script fix_markdown.sh can be used to do some checks of the Markdown documentation files.

10.4 Building the Docs

10.4.1 Purpose of this page

This page explains how this project's documentation is built and published.

For information on maintaining the *content* of documentation, see Maintaining the Docs.

10.4.2 Introduction

The majority of the documentation in ApprovalTests.cpp is maintained in Markdown format.

However, for a nicer user experience, that documentation is also generated with Sphinx and then published on Read the Docs, at approvaltestscpp.readthedocs.io.

> In most cases, it is sufficient to edit the Markdown files, and the Read the Docs site will take care of itself, as soon as changes are pushed to GitHub.

This page describes the mechanics of the documentation processes, in case they need to be worked on in future.

10.4.3 Overview

The mechanism for that publishing is based on Sy Brand's Clear, Functional C++ Documentation with Sphinx + Breathe + Doxygen + CMake.

That article gives an excellent summary of the technologies and techniques involved.

10.4.4 Required Tools

Tools:

- mdsnippets
- Doxygen
- graphviz
  - Because Doxygen will complain if it is not found
- Sphinx
- pandoc
  - "pandoc is your swiss-army knife…" for converting between markup formats
- Python3

Python3 modules:

- The required modules are defined in doc/requirements.txt
  - That can be installed by running `build/install_python_requirements.sh`
- Currently, they are:
  - breathe
    - "Breathe provides a bridge between the Sphinx and Doxygen documentation systems."
  - pypandoc
    - "Pypandoc provides a thin wrapper for pandoc, a universal document converter."
  - sphinx_rtd_theme
    - A sphinx theme, used primarily on Read the Docs

10.4.5 CMake Targets

On developer machines, where the required tools are installed, the following CMake targets are created:

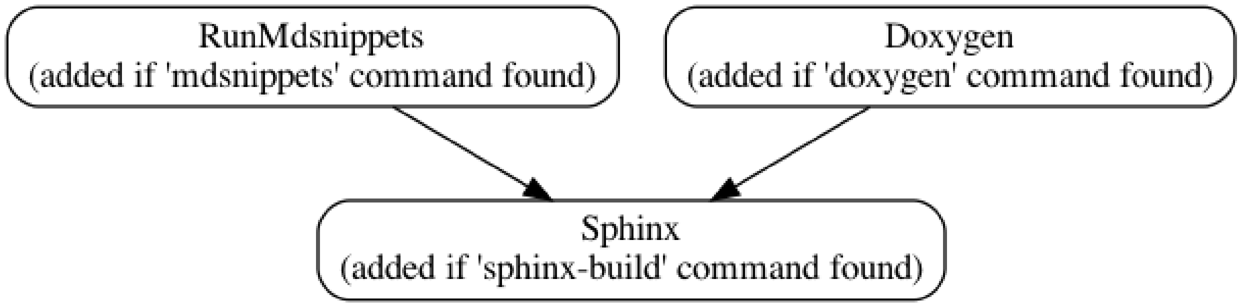

RunMdsnippets
(added if 'mdsnippets' command found)
Doxygen
(added if 'doxygen' command found)
Sphinx
(added if 'sphinx-build' command found)

Fig. 1: CMake targets for building documentation

To run all these steps, and open the Sphinx output in a web browser:

```
# in a cmake-created build tree:
cmake --build . --target Sphinx && open doc/sphinx/index.html
```

On platforms other than macOS, replace the `open` command with whatever command opens a file in a web browser.

10.4.6 About Read the Docs

- Read the Docs provides free hosting for technical documentation - most commonly generated by Sphinx.
- For this project, the documentation is at: approvaltestscpp.readthedocs.io
- The ApprovalTests.cpp Read the Docs project page has many useful links and settings.

"Read the Docs" builds

- The documentation is automatically updated whenever changes are pushed to the main branch.
- It typically takes 1 to 2 minutes for the documentation to be updated
- We currently don't get notifications for any build failures
- There is a "docs" build badge alongside our other badges on the github page, though.
- To see the build logs:
  1. click on the little green "v.latest" at the bottom of the navigation panel.
  2. click on "Builds"
  3. or click this link to the Builds.

10.4.7 Known Issues

Issues with Sphinx Output

- Images are loaded from github, rather than being copied in to the Sphinx output. This means that when editing the documentation, changes to images need to be pushed to GitHub to see the effect.
- Formatting
  - pandoc wraps long lines in tables. This results in long words in tables being broken up, for example there are spaces in some macro names in "Consistent macro names"
  - Formatting of the C++ code isn't great: for example, it would benefit from more whitespace, for readability.
- URLs
  - I am unsure about the appearance of `/generated_docs/` in the URLs of pages generated from MarkDown.
    - It avoids having to git-ignore some `.rst` files in a folder that contains a version-controlled `.rst` file
    - But it does clutter up and create longer URLs
- Links
  - In the PDF version, there are some broken links
  - Links in the API docs - that were generated from Doxygen - go to the docs on GitHub. It would be much nicer if they jumped to the relevant Sphinx page.

10.4.8 Implementation Details

The rest of this document explains the file conversion processes, in case anyone else needs to maintain them.

Images

- `doc/images/*`
  - Images for inclusion in docs
- `doc/images/tutorial/*`
  - Images for inclusion in docs
- `doc/images/source/*`
  - Sources for some of the images.
- `doc/images/source/generate_images.py`
  - Script that runs graphviz's `dot` to generate images from some source files.

Step 1: mdsnippets and Markdown Files

mdsnippets Summary

- Purpose:
  - Update the machine-generated markdown files, which will later be used as inputs to the Sphinx documentation
- Output Files:
  - `doc/*.md`
  - `doc/explanations/*.md`
  - `doc/how_tos/*.md`

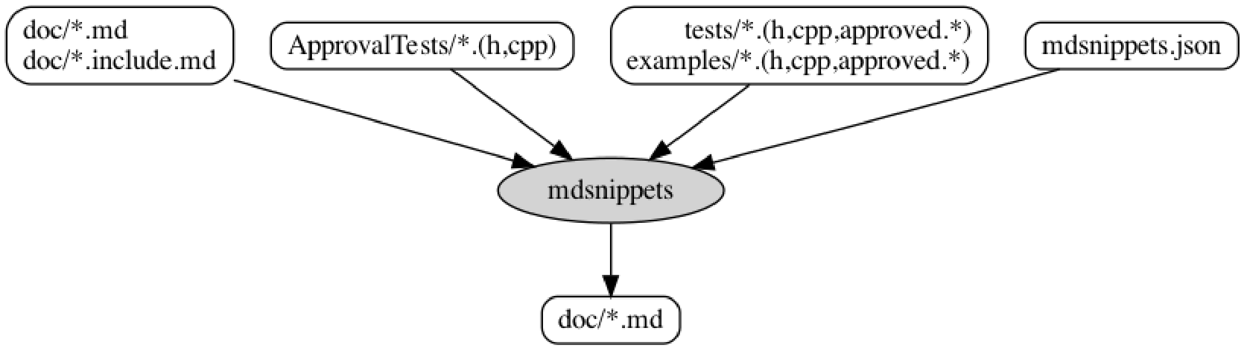


Fig. 2: Flow of Markdown files through mdsnippets

mdsnippets Details

- Configuration files:
  - `doc/run_mdsnippets/CMakeLists.txt`
    - Creates a CMake target `RunMdsnippets`
      - Note: this is not included in the target `all`
    - This target makes it convenient to quickly run `run_markdown_templates.sh` from within CLion, without switching to a console window.
  - `mdsnippets.json`
    - Configuration used by mdsnippets
- Input files:
  - See Maintaining the Docs for details.

Step 2: Doxygen conversion

Doxygen Summary

- Purpose:
  - Read the library's source code, to generate a set of XML files that describe the API
  - These XML files will later be read by Sphinx to create the API documentation
- Output Files:
  - `build_tree/doc/doxygen/xml/index.xml`
    - This and related .xml files are supplied to Sphinx, to generate the API docs on Read the Docs
  - `build_tree/doc/doxygen/html/index.html`
    - This file may be useful on developer machines, to review the documentation generated by Doxygen

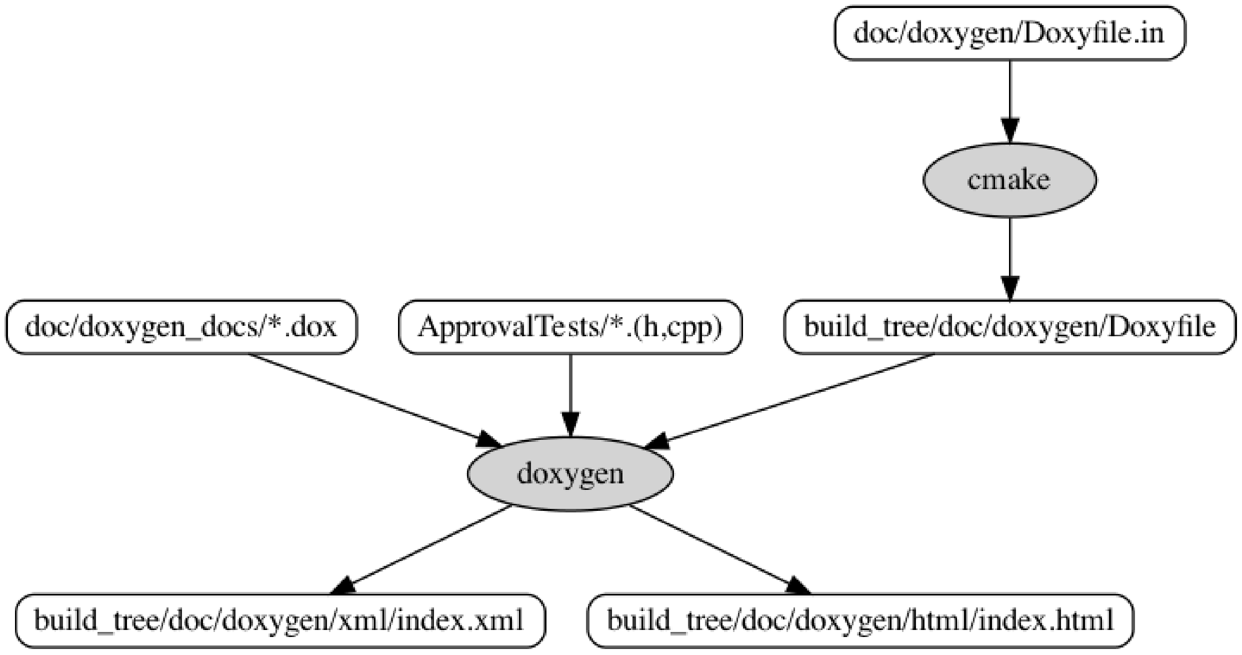

doc/doxygen/Doxyfile.in
cmake
doc/doxygen_docs/*.dox
ApprovalTests/*.(h,cpp)
build_tree/doc/doxygen/Doxyfile
doxygen
build_tree/doc/doxygen/xml/index.xml
build_tree/doc/doxygen/html/index.html

Fig. 3: Flow of files through doxygen

Doxygen Details

- Configuration files:
  - `doc/doxygen/CMakeLists.txt`
    - Creates a CMake target `Doxygen`
      - Note: this is not included in the target `all`
  - `doc/doxygen/Doxyfile.in`
    - Template file containing the Doxygen configuration
    - CMake converts it to `Doxyfile` in the build tree

- Input files:
  - `doc/doxygen_doc/*.dox`
    - Files containing descriptive text for including in the Doxygen documentation, which will then be included in the Sphinx docs.
  - `doc/ApprovalTests/*.cpp`
  - `doc/ApprovalTests/*.h`

Step3: reStructuredText and Sphinx

Sphinx Summary

- Purpose:
  - Use the Sphinx system to generate a nicely formatted, usable version of our Markdown and C++ documentation, for serving from Read the Docs
  - This involves processing and converting our Markdown files to reStructuredText format
- Output Files:
  - `build_tree/doc/sphinx/index.html`
  - Read the Docs documentation

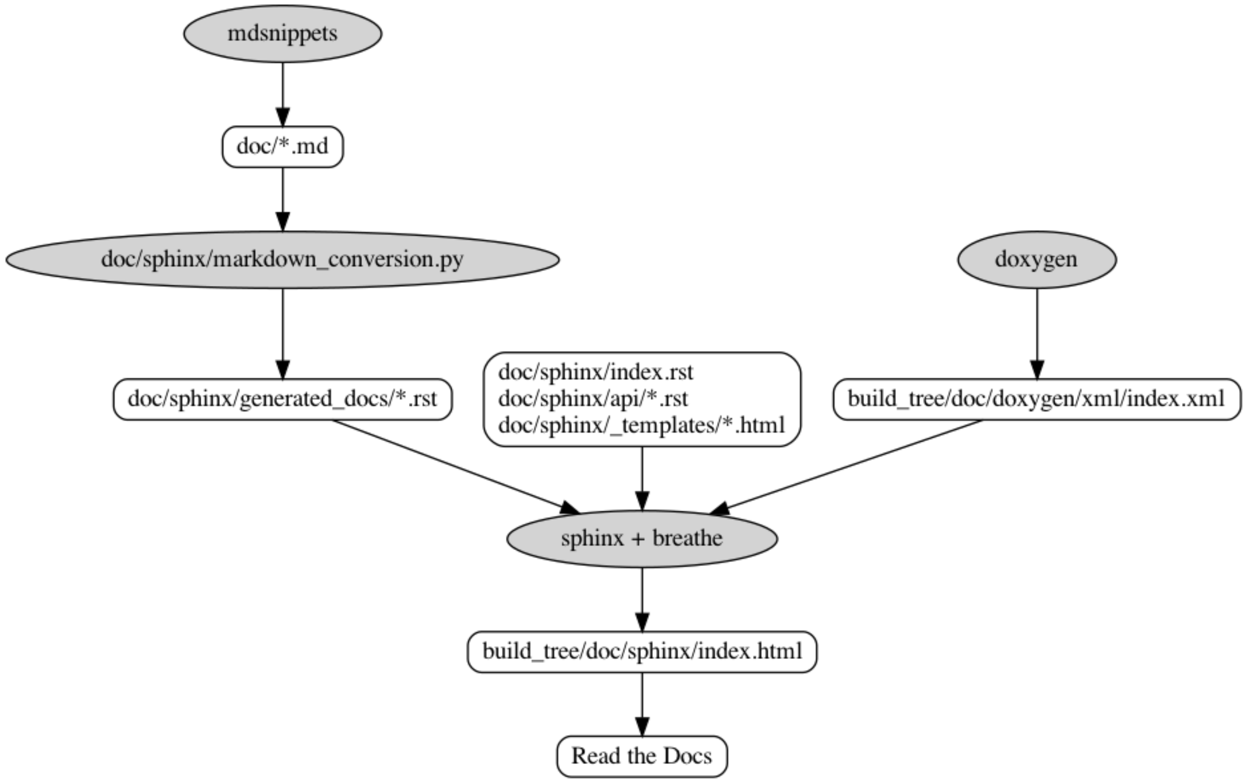

mdsnippets
doc/*.md
doc/sphinx/markdown_conversion.py
doxygen
doc/sphinx/generated_docs/*.rst
doc/sphinx/index.rst
doc/sphinx/api/*.rst
doc/sphinx/_templates/*.html
build_tree/doc/doxygen/xml/index.xml
sphinx + breathe
build_tree/doc/sphinx/index.html
Read the Docs

Fig. 4: Flow of files through Sphinx

Sphinx Details

- Configuration files:
  - `doc/requirements.txt`
    - The Python requirements for running all Python scripts in `doc/`
    - Can be installed with pip3 by running `build/install_python_requirements.sh`
  - `doc/sphinx/CMakeLists.txt`
    - Creates a CMake target `Sphinx`
      - Note: this is not included in the target `all`
  - `doc/sphinx/_templates/breadcrumbs.html`
    - Prevents a broken "Edit on GitHub" from being added to each page on Read the Docs
    - This is because most of our Sphinx Documentation pages are machine-generated, so the "Edit on GitHub" would take users to a non-existent page
- Scripts:
  - `doc/sphinx/__init__.py`
  - `doc/sphinx/conf.py`
    - This contains our Sphinx configuration
    - It also runs the script markdown_conversion.py
    - It has some extra steps that are run only when generating the docs on Read the Docs (where CMake is not available)
  - `doc/sphinx/markdown_conversion.py`
    - Convert all the `*.md` files generated by mdsnippets to `*.rst` files for feeding in to Sphinx
    - For example, it changes some links so that they go to other pages in Sphinx
    - And other links it sends to the GitHub site
- Tests:
  - `doc/sphinx/tests/test_markdown_conversion.py`
    - Unit and Approval Tests for `doc/sphinx/tests/test_markdown_conversion.py`
  - `doc/sphinx/tests/test_markdown_conversion_input.md`
    - An input file with a range of different types of Markdown constructs, taken from our own documentation
  - `doc/sphinx/tests/TestWholeConversion.test_convert_markdown_for_pandoc.approved.md`
    - To see the 1st stage of transformations made to markdown files, compare this with:
      - `doc/sphinx/tests/test_markdown_conversion_input.md`
  - `doc/sphinx/tests/TestWholeConversion.test_convert_markdown_for_pandoc.approved.rst`
    - To see how converted mardown files appear in `.rst` format, compare this with:
      - `doc/sphinx/tests/TestWholeConversion.test_convert_markdown_for_pandoc.approved.md`

- Input files:
  - `doc/sphinx/index.rst`
  - `doc/sphinx/api/*.rst`
  - Plus all the outputs from mdsnippets:
    - `doc/*.md`
    - `doc/explanations/*.md`
    - `doc/how_tos/*.md`
- Intermediate files:
  - `doc/sphinx/generated_docs/*.rst`
  - `doc/sphinx/generated_docs/explanations/*.rst`
  - `doc/sphinx/generated_docs/how_tos/*.rst`
  - These are all ignored by git

CHAPTER

ELEVEN

C++ REFERENCE

This section contains a growing list of pages documenting the ApprovalTests.cpp API. It may be useful to see what methods, and overloads, are available. It is very short on descriptive text, as we focus our efforts on the documentation above.

- **Fundamentals**: *Approving Objects* | *Core Classes* | *Scrubber Functions*

11.1 Approving Objects

Note: All classes and symbols listed here are in the `ApprovalTests` namespace.

11.1.1 Approvals

using `ApprovalTests::`**`Approvals`** = *TApprovals*<ToStringCompileTimeOptions<StringMaker>>

TApprovals

template<typename **`TCompileTimeOptions`**>

class **`TApprovals`**

Verifying single objects

See Testing Single Objects in the User Guide on GitHub.

static inline void **`verify`**(const std::string &contents, const *Options* &options = *Options*())

template<typename **`T`**, typename = *IsNotDerivedFromWriter*<*T*>>
static inline void **`verify`**(const *T* &contents, const *Options* &options = *Options*())

template<typename **`T`**, typename **`Function`**, typename = Detail::EnableIfNotOptionsOrReporter<*Function*>>
static inline void **`verify`**(const *T* &contents, *Function* converter, const *Options* &options = *Options*())

static inline void **`verify`**(const *ApprovalWriter* &writer, const *Options* &options = *Options*())
Note that this overload ignores any scrubber in options.

Verifying containers of objects - supplying an iterator range

See Testing Containers in the User Guide on GitHub.

template<typename **Iterator**>
static inline void **verifyAll**(const std::string &header, const *Iterator* &start, const *Iterator* &finish, std::function<void(decltype(**start*), std::ostream&)> converter, const *Options* &options = *Options*())

Verifying containers of objects - supplying a container

See Testing Containers in the User Guide on GitHub.

template<typename **Container**>
static inline void **verifyAll**(const std::string &header, const *Container* &list, std::function<void(typename *Container*::value_type, std::ostream&)> converter, const *Options* &options = *Options*())

template<typename **Container**>
static inline void **verifyAll**(const std::string &header, const *Container* &list, const *Options* &options = *Options*())

template<typename **Container**>
static inline void **verifyAll**(const *Container* &list, const *Options* &options = *Options*())

Verifying containers of objects - supplying an initializer list

See Testing Containers in the User Guide on GitHub.

template<typename **T**>
static inline void **verifyAll**(const std::string &header, const std::initializer_list<*T*> &list, std::function<void(typename std::initializer_list<*T*>::value_type, std::ostream&)> converter, const *Options* &options = *Options*())

template<typename **T**>
static inline void **verifyAll**(const std::string &header, const std::initializer_list<*T*> &list, const *Options* &options = *Options*())

template<typename **T**>
static inline void **verifyAll**(const std::initializer_list<*T*> &list, const *Options* &options = *Options*())

Other verify methods

static inline void **verifyExceptionMessage**(const std::function<void(void)> &functionThatThrows, const *Options* &options = *Options*())

Verify the text of an exception.

See Testing exception messages in the User Guide on GitHub.

static inline void **verifyExistingFile**(const std::string &filePath, const *Options* &options = *Options*())

Verify an existing file, that has already been written out.

Customising Approval Tests

These static methods customise various aspects of Approval Tests behaviour.

static inline SubdirectoryDisposer **useApprovalsSubdirectory**(const std::string &subdirectory = "approval_tests")

See Using sub-directories for approved files in the User Guide on GitHub.

static inline DefaultReporterDisposer **useAsDefaultReporter**(const std::shared_ptr<*Reporter*> &reporter)

See Registering a default reporter in the User Guide on GitHub.

static inline FrontLoadedReporterDisposer **useAsFrontLoadedReporter**(const std::shared_ptr<*Reporter*> &reporter)

See Front Loaded Reporters in the User Guide on GitHub.

static inline DefaultNamerDisposer **useAsDefaultNamer**(NamerCreator namerCreator)

See Registering a Custom Namer in the User Guide on GitHub.

static inline FileNameSanitizerDisposer **useFileNameSanitizer**(FileNameSanitizer sanitizer)

See Converting Test Names to Valid FileNames in the User Guide on GitHub.

Public Types

template<typename **T**>

using **IsNotDerivedFromWriter** = typename std::enable_if<!std::is_base_of<*ApprovalWriter*, *T*>::value, int>::type

Public Static Functions

static inline std::shared_ptr<*ApprovalNamer*> **getDefaultNamer**()

11.1.2 CombinationApprovals

using ApprovalTests::**CombinationApprovals** = *TCombinationApprovals*<ToStringCompileTimeOptions<APPROVAL_TESTS_DEFAULT_STREAM_CONVERTER>>

TCombinationApprovals

template<typename **`TCompileTimeOptions`**>

class **`TCombinationApprovals`**

Verifying combinations of objects

See Testing combinations in the User Guide on GitHub.

template<class **`Converter`**, class **`Container`**, class ...**`Containers`**>
static inline void **`verifyAllCombinations`**(const *Options* &options, const std::string &header, *Converter* &&converter, const *Container* &input0, const *Containers*&... inputs)

template<class **`Converter`**, class ...**`Containers`**>
static inline ApprovalTests::Detail::EnableIfNotOptionsOrReporterOrString<*Converter*> **`verifyAllCombinations`**(const std::string &header, *Converter* &&converter, const *Containers*&... inputs)

template<class **`Converter`**, class ...**`Containers`**>
static inline ApprovalTests::Detail::EnableIfNotOptionsOrReporterOrString<*Converter*> **`verifyAllCombinations`**(const *Options* &options, *Converter* &&converter, const *Containers*&... inputs)

template<class **`Converter`**, class ...**`Containers`**>

static inline ApprovalTests::Detail::EnableIfNotOptionsOrReporterOrString<*Converter*> **`verifyAllCombinations`**`(`*Converter* &&converter, const *Containers*&... inputs`)`

11.2 Core Classes

Note: All classes and symbols listed here are in the `ApprovalTests` namespace.

11.2.1 ApprovalComparator

`ApprovalComparator` is an interface that determines if received and approved files are equivalent.

For more information, see *Custom Comparators*.

class **`ApprovalComparator`**

Subclassed by TextFileComparator

Public Functions

virtual **`~ApprovalComparator`**`()` = default

virtual bool **`contentsAreEquivalent`**`(`std::string receivedPath, std::string approvedPath`)` const = 0

11.2.2 ApprovalException and subclasses

`ApprovalException` is the base class for ApprovalTest-specific exceptions.

class **`ApprovalException`** : public exception

Subclassed by *ApprovalMismatchException*, *ApprovalMissingException*

Public Functions

explicit **`ApprovalException`**(const std::string &msg)

virtual const char ***`what`**() const noexcept override

ApprovalMismatchException

`ApprovalMismatchException` is thrown if received and approved files differ.

class **`ApprovalMismatchException`** : public *ApprovalException*

Public Functions

`ApprovalMismatchException`(const std::string &received, const std::string &approved)

ApprovalMissingException

`ApprovalMissingException` is thrown if the approved file is missing - typically on first run of a new test.

class **`ApprovalMissingException`** : public *ApprovalException*

Public Functions

`ApprovalMissingException`(const std::string&, const std::string &approved)

11.2.3 ApprovalNamer

`ApprovalNamer` is the interface that controls how approved and received files are named.

For more information, see *Namers*.

class **`ApprovalNamer`**
Subclassed by ApprovalTestNamer, ExistingFileNamer, TemplatedCustomNamer

Public Functions

virtual **`~ApprovalNamer`**() = default

virtual std::string **`getApprovedFile`**(std::string extensionWithDot) const = 0

virtual std::string **`getReceivedFile`**(std::string extensionWithDot) const = 0

11.2.4 ApprovalWriter

`ApprovalWriter` is the interface that controls how objects being verified are written to disk.

For more information, see *Writers*.

class **ApprovalWriter**
Subclassed by ExistingFile, StringWriter

Public Functions

virtual **~ApprovalWriter**() = default

virtual std::string **getFileExtensionWithDot**() const = 0

virtual void **write**(std::string path) const = 0

virtual void **cleanUpReceived**(std::string receivedPath) const = 0

11.2.5 FileApprover

Low-level methods for approving files.

class **FileApprover**

Public Types

using **TestPassedNotification** = std::function<void()>

Public Functions

FileApprover() = default

~FileApprover() = default

Public Static Functions

static ComparatorDisposer **registerComparatorForExtension**(const std::string &extensionWithDot, std::shared_ptr<*ApprovalComparator*> comparator)

Register a custom comparater, which will be used to compare approved and received files with the given extension.

See also:

For more information, see Custom Comparators in the User Guide on GitHub.

Parameters

- `extensionWithDot` – A file extention, such as ".jpg"
- `comparator` – `std::shared_ptr` to a *ApprovalTests::ApprovalComparator* instance

Returns A "Disposable" object. The caller should hold on to this object. When it is destroyed, the customisation will be reversed.

static void **`verify`**(const std::string &receivedPath, const std::string &approvedPath, const *ApprovalComparator* &comparator)

This overload is an implementation detail. To add a new comparator, use *registerComparatorForExtension()*.

static void **`verify`**(const std::string &receivedPath, const std::string &approvedPath)

static void **`verify`**(const *ApprovalNamer* &n, const *ApprovalWriter* &s, const *Reporter* &r)

static void **`reportAfterTryingFrontLoadedReporter`**(const std::string &receivedPath, const std::string &approvedPath, const *Reporter* &r)

static void **`setTestPassedNotification`**(*TestPassedNotification* notification)

static void **`notifyTestPassed`**()

11.2.6 Options

Easy control of various customization points in Approvals::verify() and similar methods

For more information, see *Options*.

class **`Options`**

Public Functions

`Options`() = default

explicit **`Options`**(Scrubber scrubber)

explicit **`Options`**(const *Reporter* &reporter)

FileOptions **`fileOptions`**() const

Scrubber **`getScrubber`**() const

bool **isUsingDefaultScrubber**() const

std::string **scrub**(const std::string &input) const

const *Reporter* &**getReporter**() const

Options **withReporter**(const *Reporter* &reporter) const

Options **withScrubber**(Scrubber scrubber) const

std::shared_ptr<*ApprovalNamer*> **getNamer**() const

Options **withNamer**(std::shared_ptr<*ApprovalNamer*> namer)

class **FileOptions**

Public Functions

const std::string &**getFileExtension**() const

Options **withFileExtension**(const std::string &fileExtensionWithDot) const

11.2.7 Reporter and subclasses

Reporters are called on test failure, typically to show differences.

Most reporters launch an external diffing tool, allowing programmers to understand the difference(s) between approved and received files. They also typically allow the output to be "approved".

For a demonstration of this, see the *Tutorial*.

For more information, see *Reporters*.

class **Reporter**

Subclassed by AutoApproveIfMissingReporter, AutoApproveReporter, BlockingReporter, CIBuildOnlyReporter, ClipboardReporter, CombinationReporter, CommandReporter, DefaultReporter, EnvironmentVariableReporter, FirstWorkingReporter, QuietReporter, TextDiffReporter

Public Functions

virtual **`~Reporter`**`()` = default

virtual bool **`report`**`(`std::string received, std::string approved`)` const = 0

Linux Reporters

namespace **`Linux`**

class **`SublimeMergeSnapReporter`** : public GenericDiffReporter
#include <LinuxReporters.h>

class **`SublimeMergeFlatpakReporter`** : public GenericDiffReporter
#include <LinuxReporters.h>

class **`SublimeMergeRepositoryPackageReporter`** : public GenericDiffReporter
#include <LinuxReporters.h>

class **`SublimeMergeDirectDownloadReporter`** : public GenericDiffReporter
#include <LinuxReporters.h>

class **`SublimeMergeReporter`** : public FirstWorkingReporter
#include <LinuxReporters.h>

class **`KDiff3Reporter`** : public GenericDiffReporter
#include <LinuxReporters.h>

class **`MeldReporter`** : public GenericDiffReporter
#include <LinuxReporters.h>

class **`BeyondCompareReporter`** : public GenericDiffReporter
#include <LinuxReporters.h>

class **`LinuxDiffReporter`** : public FirstWorkingReporter
#include <LinuxReporters.h>

macOS Reporters

namespace **`Mac`**

class **`DiffMergeReporter`** : public GenericDiffReporter
#include <MacReporters.h>

class **`AraxisMergeReporter`** : public GenericDiffReporter
#include <MacReporters.h>

class `VisualStudioCodeReporter` : public GenericDiffReporter
#include <MacReporters.h>

class `BeyondCompareReporter` : public GenericDiffReporter
#include <MacReporters.h>

class `KaleidoscopeReporter` : public GenericDiffReporter
#include <MacReporters.h>

class `SublimeMergeReporter` : public GenericDiffReporter
#include <MacReporters.h>

class `KDiff3Reporter` : public GenericDiffReporter
#include <MacReporters.h>

class `P4MergeReporter` : public GenericDiffReporter
#include <MacReporters.h>

class `TkDiffReporter` : public GenericDiffReporter
#include <MacReporters.h>

class `CLionDiffReporter` : public GenericDiffReporter
#include <MacReporters.h>

class `MacDiffReporter` : public FirstWorkingReporter
#include <MacReporters.h>

Windows Reporters

namespace `Windows`

class `VisualStudioCodeReporter` : public GenericDiffReporter
#include <WindowsReporters.h>

class `BeyondCompare3Reporter` : public GenericDiffReporter
#include <WindowsReporters.h>

class `BeyondCompare4Reporter` : public GenericDiffReporter
#include <WindowsReporters.h>

class `BeyondCompareReporter` : public FirstWorkingReporter
#include <WindowsReporters.h>

class `TortoiseImageDiffReporter` : public GenericDiffReporter
#include <WindowsReporters.h>

class `TortoiseTextDiffReporter` : public GenericDiffReporter
#include <WindowsReporters.h>

class **`TortoiseDiffReporter`** : public FirstWorkingReporter
#include <WindowsReporters.h>

class **`TortoiseGitTextDiffReporter`** : public GenericDiffReporter
#include <WindowsReporters.h>

class **`TortoiseGitImageDiffReporter`** : public GenericDiffReporter
#include <WindowsReporters.h>

class **`TortoiseGitDiffReporter`** : public FirstWorkingReporter
#include <WindowsReporters.h>

class **`WinMergeReporter`** : public GenericDiffReporter
#include <WindowsReporters.h>

class **`AraxisMergeReporter`** : public GenericDiffReporter
#include <WindowsReporters.h>

class **`CodeCompareReporter`** : public GenericDiffReporter
#include <WindowsReporters.h>

class **`SublimeMergeReporter`** : public GenericDiffReporter
#include <WindowsReporters.h>

class **`KDiff3Reporter`** : public GenericDiffReporter
#include <WindowsReporters.h>

class **`WindowsDiffReporter`** : public FirstWorkingReporter
#include <WindowsReporters.h>

11.3 Scrubber Functions

Note: All classes and symbols listed here are in the `ApprovalTests` namespace.

11.3.1 Scrubbers

For more information, see *Scrubbers*.

namespace **`Scrubbers`**

Regex-based scrubbers

See Regular Expressions (regex) in the User Guide on GitHub.

using **RegexMatch** = std::sub_match<std::string::const_iterator>

using **RegexReplacer** = std::function<std::string(const *RegexMatch*&)>

std::string **scrubRegex**(const std::string &input, const std::regex &regex, const *RegexReplacer* &replaceFunction)

Scrubber **createRegexScrubber**(const std::regex &regexPattern, const *RegexReplacer* &replacer)

Scrubber **createRegexScrubber**(const std::regex &regexPattern, const std::string &replacementText)

Scrubber **createRegexScrubber**(const std::string &regexString, const std::string &replacementText)

Functions

std::string **doNothing**(const std::string &input)

std::string **scrubGuid**(const std::string &input)

CHAPTER

TWELVE

CODE INDEX

- genindex

INDEX