Mistakes or opportunities…

TESTING C++ CODE AN INTRODUCTION

Software Design

Louis Langholtz

ONLINE

OVERVIEW

- Speakers.
- Meetup group.
- Fostering community.
- Giving back.
- Passion.
- Opinion.
- Experience:
  - Waste no crisis!

WHAT?

CONCEPT OF WHAT?

- Relational consistencies.
  - Assignment -> equality.
  - Cause -> effect.
- Assumptions.

PERHAPS...

- **Type traits.**
- **Contracts/documentation/reasonable: explicit, implicit.**
- **Test code paths not yet covered.**
- **Readability (code-review).**
- **Less virtualizable things. Ex: power consumption, speed.**
- **Others?**

BUT NOT?

- Code to test code.
- Undefined behavior:
  - Specters/ghosts/anomalies.
- Test coverage: more test code, or less production code.
- Other things?

WHEN?

AVOID BUGS ASAP

Premise: sooner detected, less expensive to fix!

DEVELOPMENT, IN STAGES

- **Write software.**
- **Unit test it.**
- **System test it.**
- **Customer testing.**

DEVELOPMENT, IN STAGES

- **Write software. Encode ideas. Review. Test at compile time!**
- **Unit test it. Code to test API at run time.**
- **System test it. Whole system correctness/performance/etc. at run time.**
- **Customer testing. ~~Suckers?~~ Too late?**

COMPILE TIME?

COMPILE TIME TESTING

- static_assert things like type traits.
- Strong types like boost units.
- Ideally, everything. Usually, only somethings.

PLEASE REJECT VOID PARAMETER-LESS FUNCTIONS.

PLEASE <u>REJECT</u> UN-SPECIFIED BEHAVIOR

PREFER PURE FUNCTIONS

- Ex C++20: auto square(auto t) { return t * t; }
- C++ Core Guideline F.8: Prefer pure functions: “easier to reason about, sometimes easier to optimize (and even parallelize), and sometimes can be memoized”.
- Impure functions harder to test. Ex: myclass::doit(int foo);
- Local reasoning, instead of remote.

IDIOMATIC?

IDIOMATIC CONSIDERATIONS

- void init(); void do_sth(); void deinit();
- What do they do?
- When do we use them?
- How often?

IDIOMATIC CONSIDERATIONS

- void init(); void do_sth(); void deinit();
- What do they do?
- When do we use them?
- How often?

- lpt init(); void do_sth(lpt); void deinit(lpt);
- Know do_sth, deinit callable after init.
- See proof-types.
- Or C++ constructor?

ASTONISHMENT?

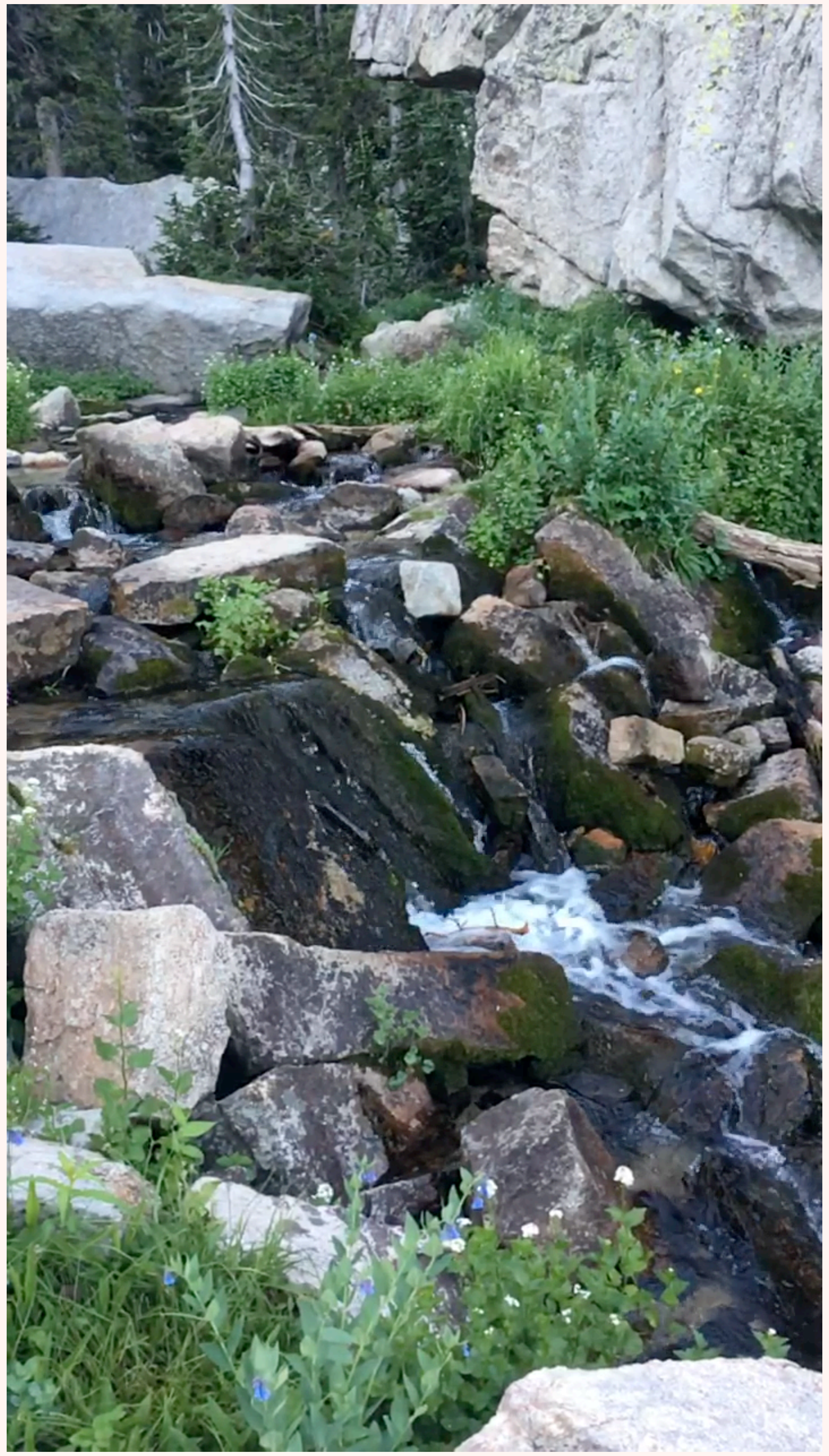
POLA FTW!

PRINCIPAL OF LEAST ASTONISHMENT

- A.K.A. “POLA”.
- “FTW” - For The Win!
- “a component of a system should behave in a way that most users will expect it to behave”
- C++ constructor most idiomatic initializer.
- C++ destructor most idiomatic de-initializer.
- Beyond that, be like int.

REGULARITY

MOUNTAIN STREAM



FOR FREE IN C++!

- “Special” member functions.
- Copy/move construction and assignment for free!
- Generated automatically.
- So our types are like int. Expectations of int. More library support.
- Unless we work against the language!

RUN TIME?

UNIT TESTS?

FRAMEWORKS

- Many available including rolling your own.
- Google test.
- Catch 2.

STYLES

- Ad-hoc.
- Fatal asserts for non-starters, non-fatal otherwise.
- AAA - Arrange, Act, Assert.

EX: FUNCTION...

```
auto square(auto t) { return t * t; }
```

EX: POD V. GET/SET…

```
struct foo {
  int a{};
  float b{};
};
```

```
struct bar {
  int get_a() const;
  float get_b() const;
  void set_a(int v);
  void set_b(float v);
private:
  int a{};
  float b{};
};
```

EX: FILE CLASS...

```
class myfile {
  int fd{-1};
  string name;
public:
  myfile() = default;
  ~myfile();
  bool is_open() const;
  string get_name() const;
  string read();
  void write(string data);
  void close() noexcept;
  void open(string name);
};
```

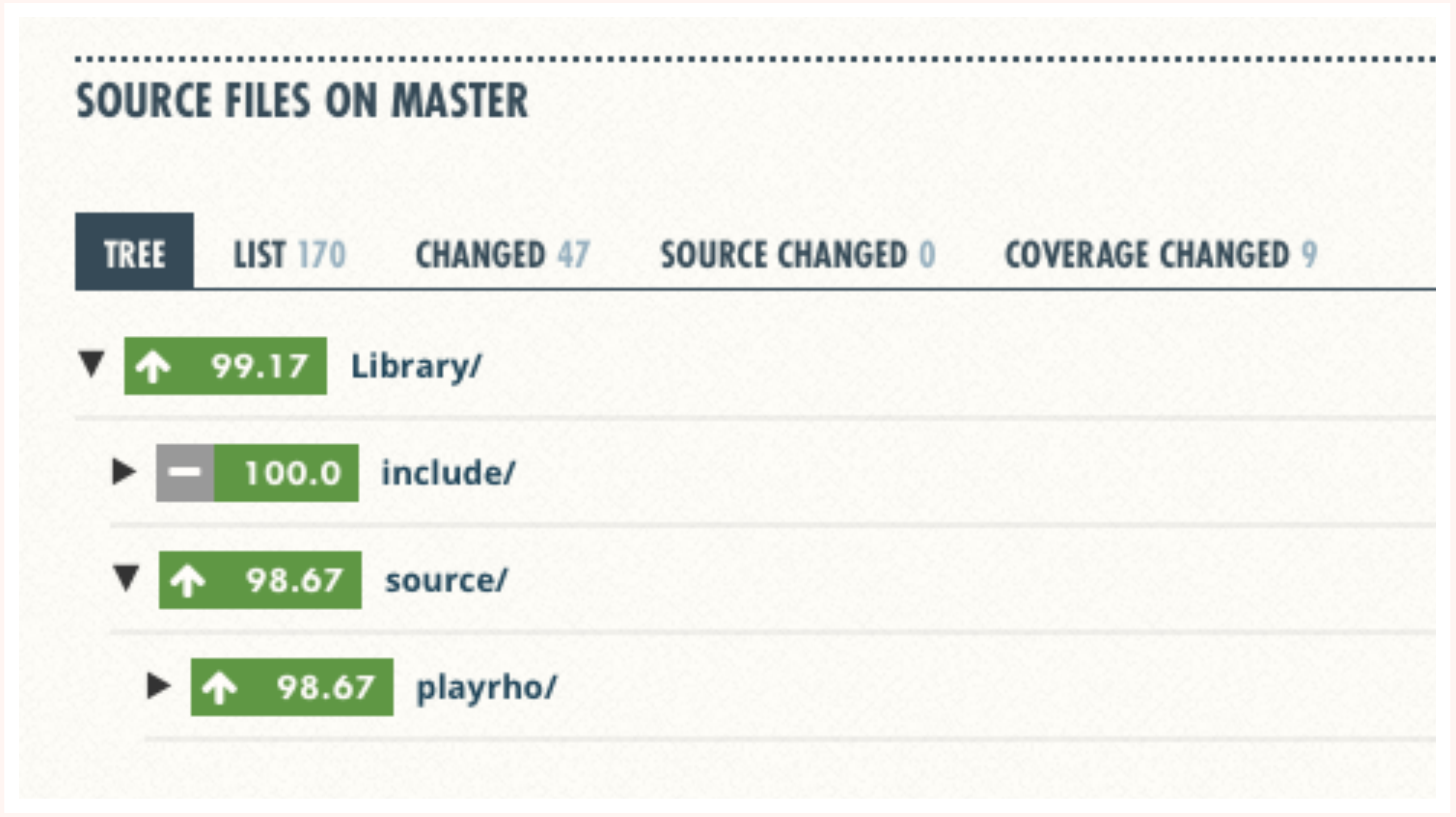
% COVERAGE?

MEASURING %

- GCC option: —coverage.
- lcov or gcov.
- Visualize: coveralls.io
- Gamified!

COVERALLS.IO

- An example...


SOURCE FILES ON MASTER

▼ ↑ 99.17 Library/
▶ − 100.0 include/
▼ ↑ 98.67 source/
▶ ↑ 98.67 playrho/

100%!

- Better as a goal - than a reality?
- Can’t test undefined behavior!
- Can’t test unreachable code!
- Harder to test intricate code.
- Harder to test remote behaviors.

- GCC option: —coverage.
- lcov or gcov.
- Visualize: coveralls.io

NO UNREACHABLE CODE.

GOOGLE TEST...

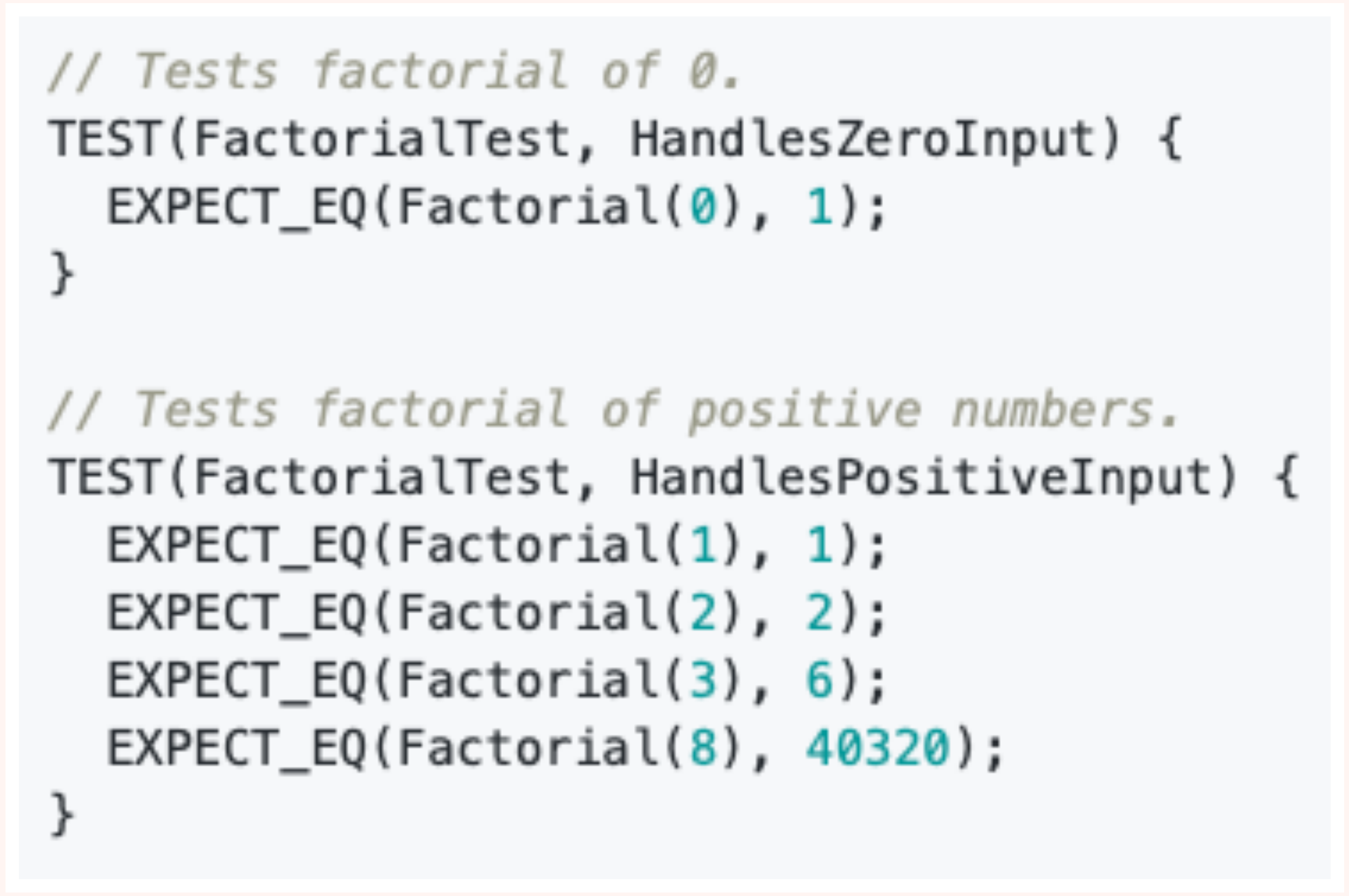
FROM GOOGLE

- **Code at: https://github.com/google/googletest**
- **Docs at: https://google.github.io/googletest/**
- **CMake: in CMakeLists.txt...**

```
include(FetchContent)
FetchContent_Declare(
  googletest
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG 391ce627def20c1e8a54d10b12949b15086473dd
)
FetchContent_MakeAvailable(googletest)
include(GoogleTest)
gtest_discover_tests(YourExecutableTargetName)
```

SIMPLE TESTS

- **Mostly what I use.**
- **TEST macro for function.**
- **Assertions: EXPECT_*, ASSERT_*.**

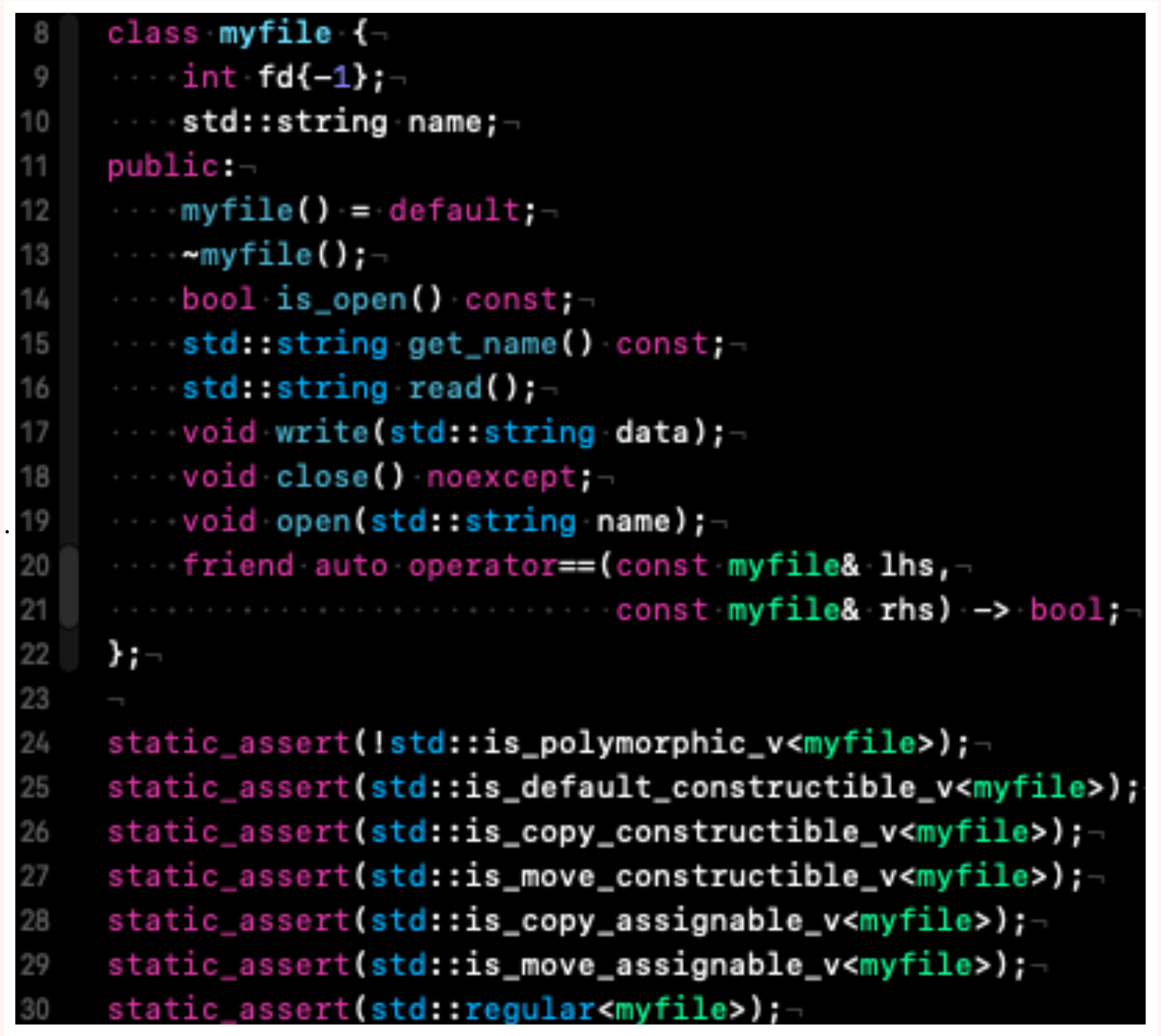


```
// Tests factorial of 0.
TEST(FactorialTest, HandlesZeroInput) {
  EXPECT_EQ(Factorial(0), 1);
}

// Tests factorial of positive numbers.
TEST(FactorialTest, HandlesPositiveInput) {
  EXPECT_EQ(Factorial(1), 1);
  EXPECT_EQ(Factorial(2), 2);
  EXPECT_EQ(Factorial(3), 6);
  EXPECT_EQ(Factorial(8), 40320);
}
```

MYFILE HEADER

- In “myfile.hpp”.
- #include <concepts>
- #include <string>
- #include <type_traits>

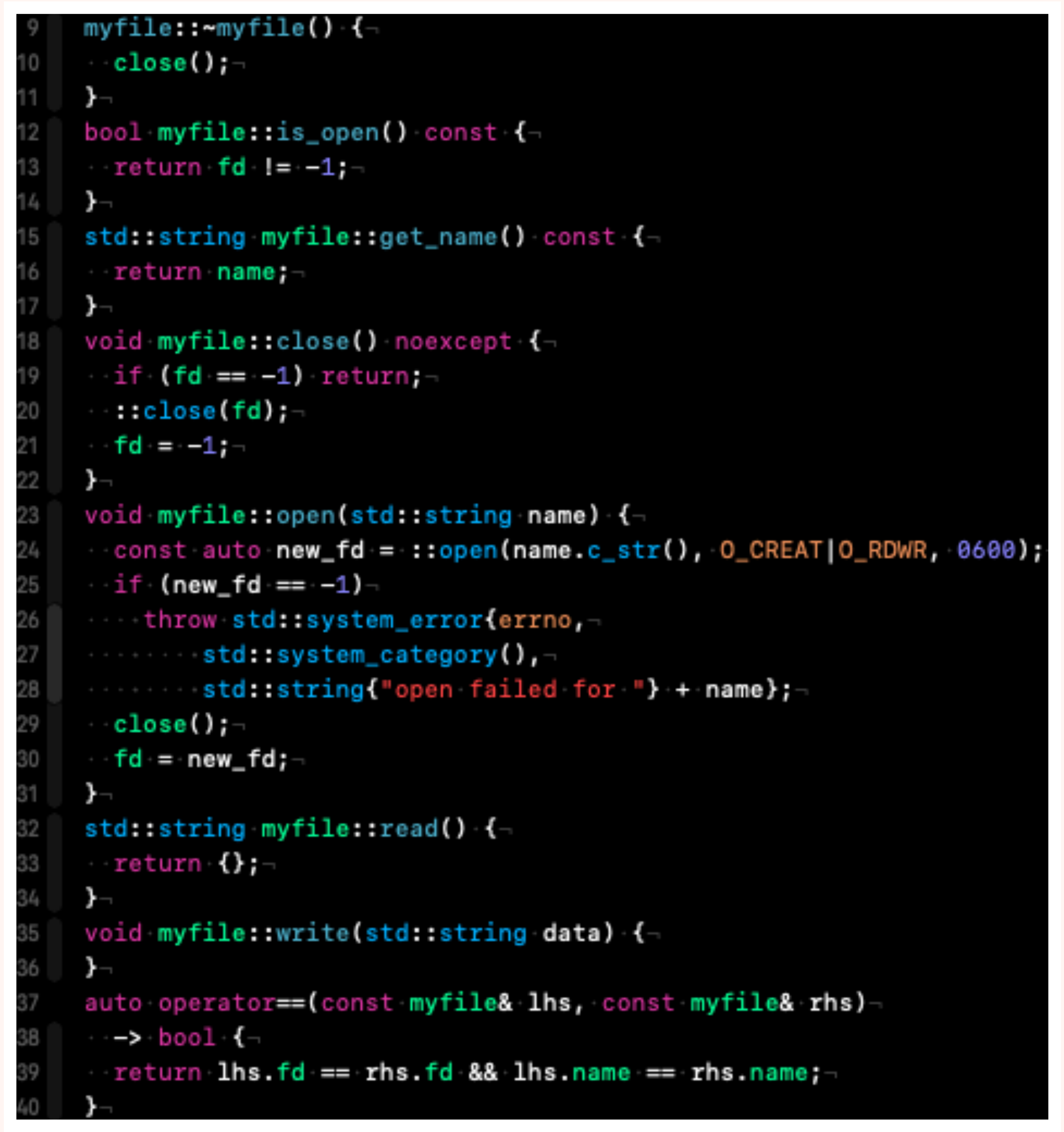


```
class myfile {
    int fd{-1};
    std::string name;
public:
    myfile() = default;
    ~myfile();
    bool is_open() const;
    std::string get_name() const;
    std::string read();
    void write(std::string data);
    void close() noexcept;
    void open(std::string name);
    friend auto operator==(const myfile& lhs,
                           const myfile& rhs) -> bool;
};

static_assert(!std::is_polymorphic_v<myfile>);
static_assert(std::is_default_constructible_v<myfile>);
static_assert(std::is_copy_constructible_v<myfile>);
static_assert(std::is_move_constructible_v<myfile>);
static_assert(std::is_copy_assignable_v<myfile>);
static_assert(std::is_move_assignable_v<myfile>);
static_assert(std::regular<myfile>);
```

MYFILE SOURCE

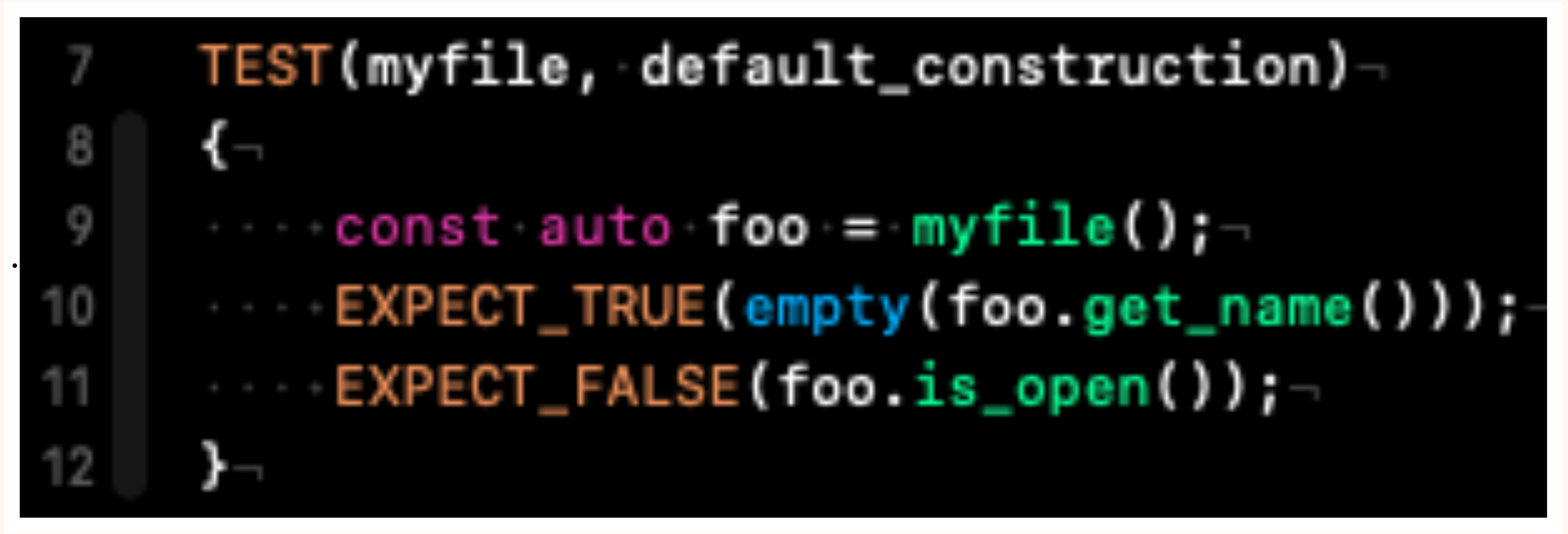
- In “myfile.cpp”.
- #include <fcntl.h>
- #include <unistd.h>
- #include <cerrno>
- #include <system_error>
- #include "myfile.hpp"



```cpp
myfile::~myfile() {
  close();
}
bool myfile::is_open() const {
  return fd != -1;
}
std::string myfile::get_name() const {
  return name;
}
void myfile::close() noexcept {
  if (fd == -1) return;
  ::close(fd);
  fd = -1;
}
void myfile::open(std::string name) {
  const auto new_fd = ::open(name.c_str(), O_CREAT|O_RDWR, 0600);
  if (new_fd == -1)
    throw std::system_error{errno,
        std::system_category(),
        std::string{"open failed for "} + name};
  close();
  fd = new_fd;
}
std::string myfile::read() {
  return {};
}
void myfile::write(std::string data) {
}
auto operator==(const myfile& lhs, const myfile& rhs)
  -> bool {
  return lhs.fd == rhs.fd && lhs.name == rhs.name;
}
```

MYFILE TESTS

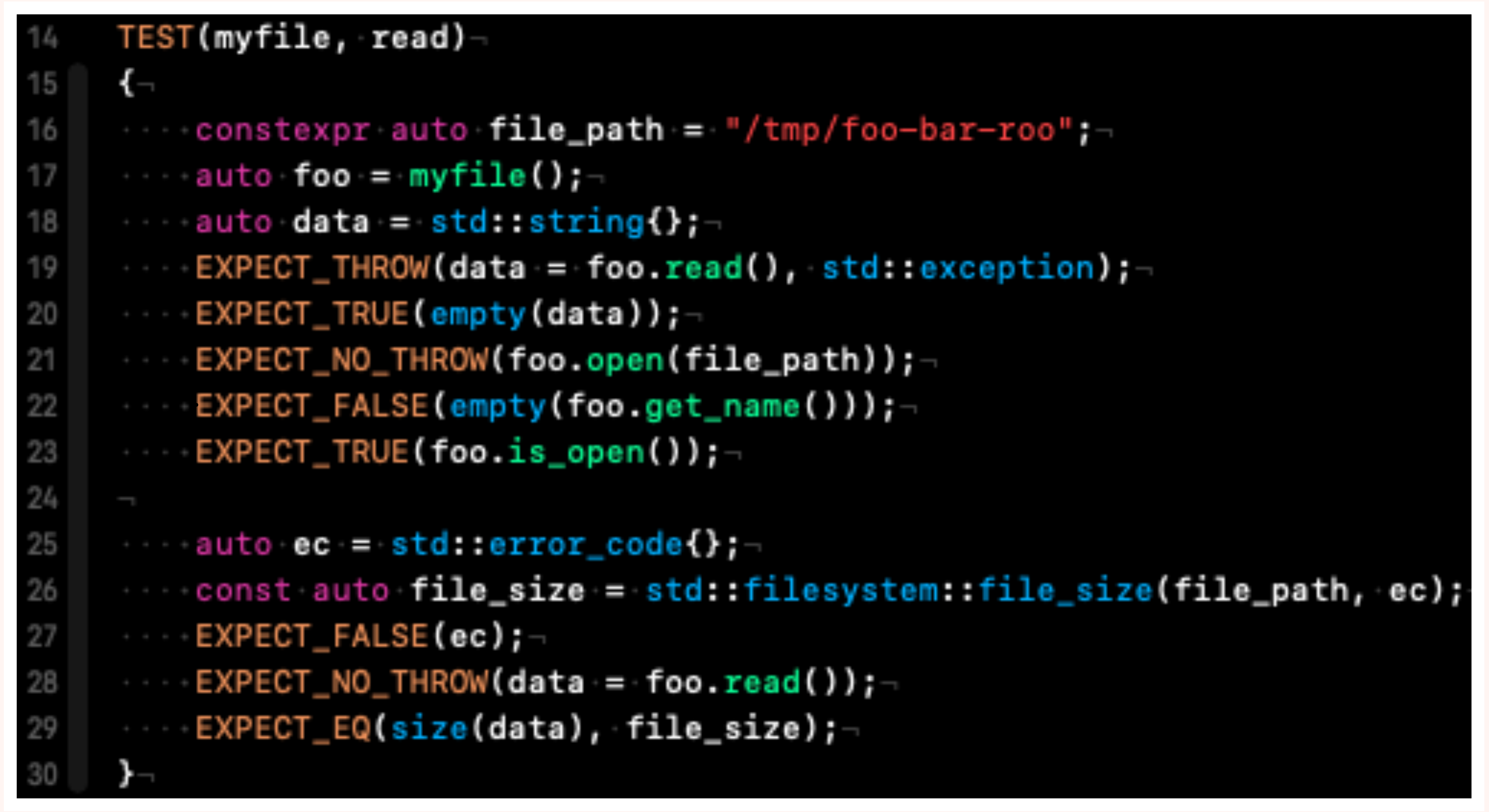
- In a “myfile.cpp” file.
- #include <gtest/gtest.h>
- #include "../library/myfile.hpp"
- TEST(myfile, default_construction)



```
TEST(myfile, default_construction)
{
    const auto foo = myfile();
    EXPECT_TRUE(empty(foo.get_name()));
    EXPECT_FALSE(foo.is_open());
}
```

MYFILE TESTS

- In a “myfile.cpp” file.
- #include <gtest/gtest.h>
- #include "../library/myfile.hpp"
- TEST(myfile, read)

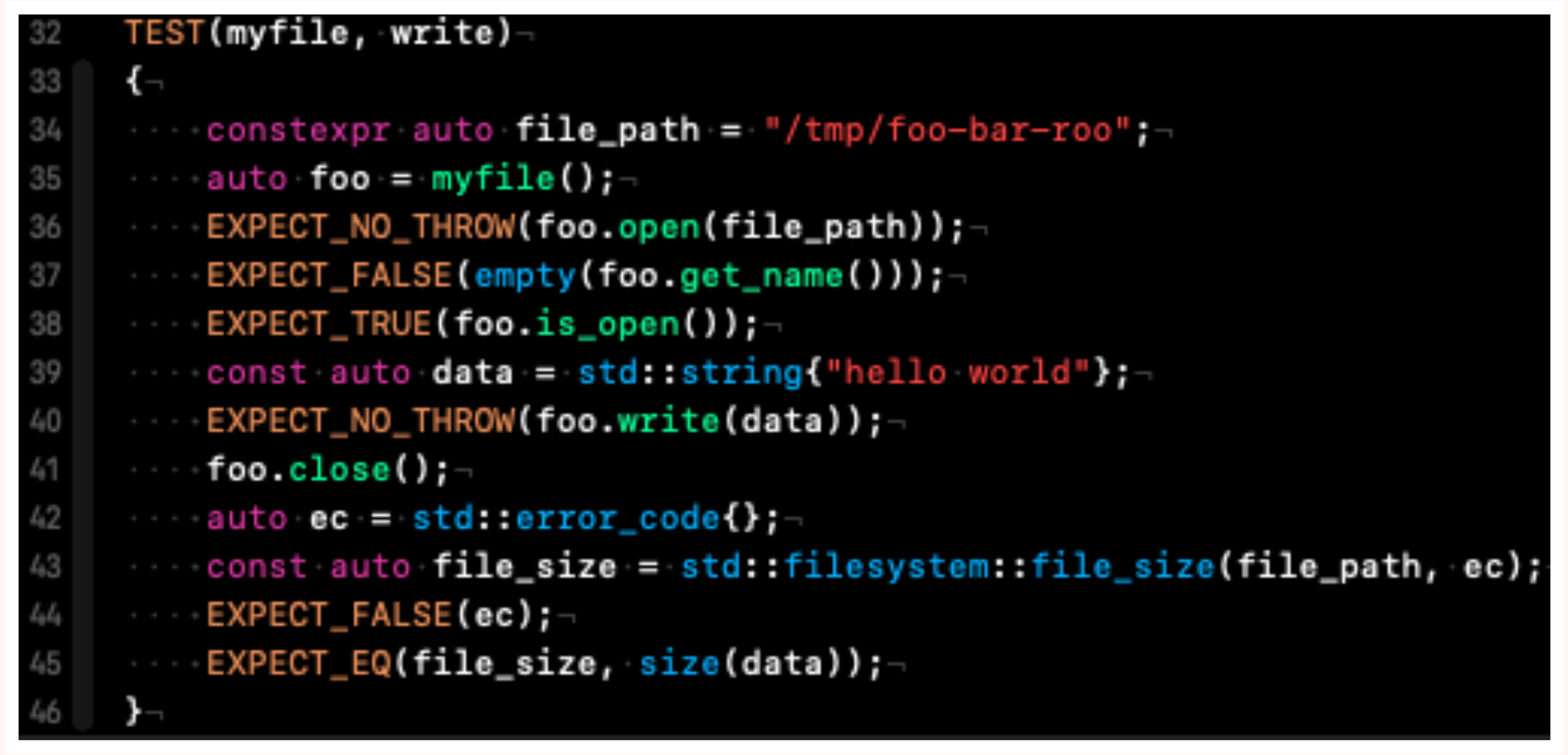


```cpp
TEST(myfile, read)
{
    constexpr auto file_path = "/tmp/foo-bar-roo";
    auto foo = myfile();
    auto data = std::string{};
    EXPECT_THROW(data = foo.read(), std::exception);
    EXPECT_TRUE(empty(data));
    EXPECT_NO_THROW(foo.open(file_path));
    EXPECT_FALSE(empty(foo.get_name()));
    EXPECT_TRUE(foo.is_open());

    auto ec = std::error_code{};
    const auto file_size = std::filesystem::file_size(file_path, ec);
    EXPECT_FALSE(ec);
    EXPECT_NO_THROW(data = foo.read());
    EXPECT_EQ(size(data), file_size);
}
```

MYFILE TESTS

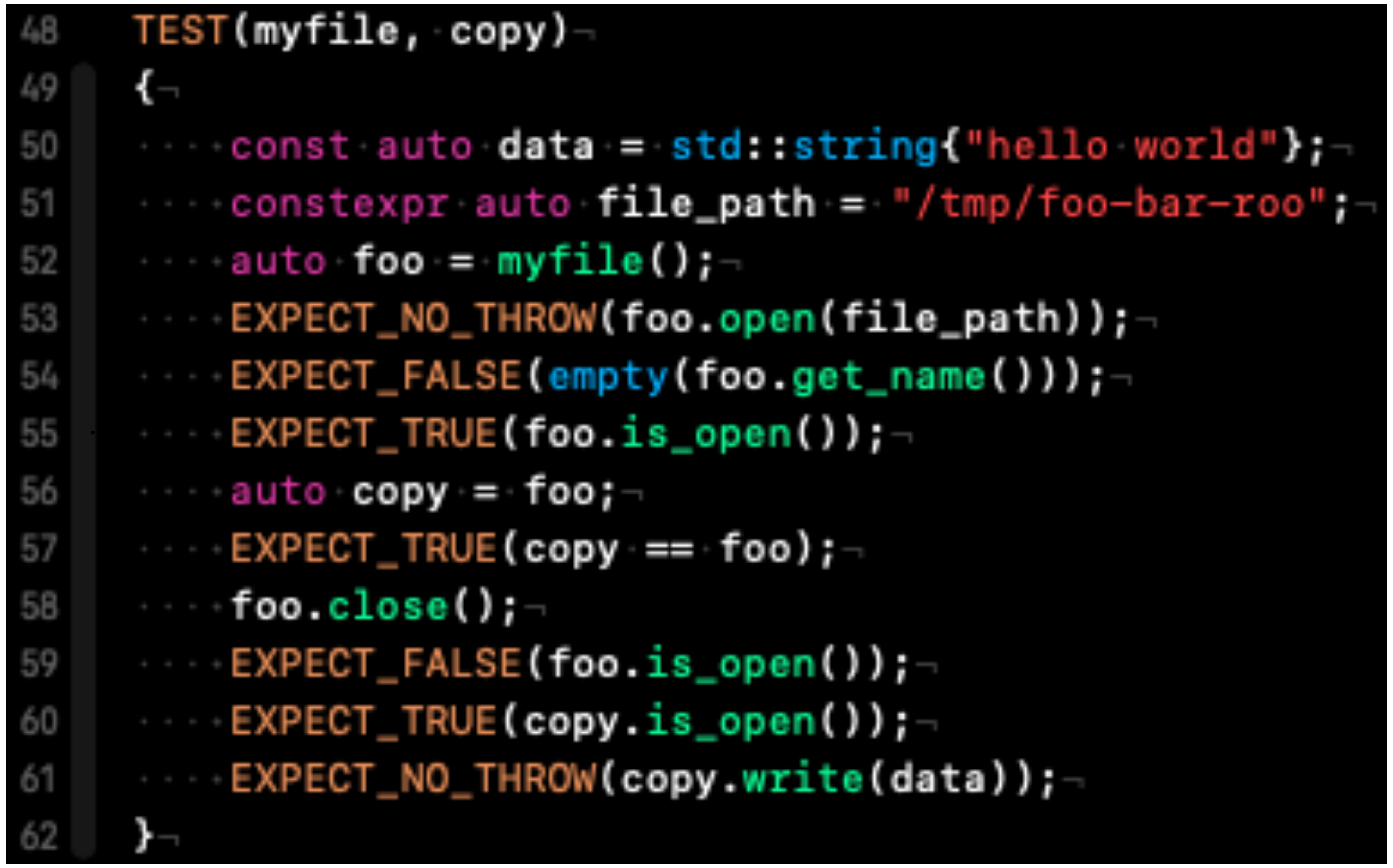
- In a “myfile.cpp” file.
- #include <gtest/gtest.h>
- #include "../library/myfile.hpp"
- TEST(myfile, write)



```cpp
TEST(myfile, write)
{
    constexpr auto file_path = "/tmp/foo-bar-roo";
    auto foo = myfile();
    EXPECT_NO_THROW(foo.open(file_path));
    EXPECT_FALSE(empty(foo.get_name()));
    EXPECT_TRUE(foo.is_open());
    const auto data = std::string{"hello world"};
    EXPECT_NO_THROW(foo.write(data));
    foo.close();
    auto ec = std::error_code{};
    const auto file_size = std::filesystem::file_size(file_path, ec);
    EXPECT_FALSE(ec);
    EXPECT_EQ(file_size, size(data));
}
```

MYFILE TESTS

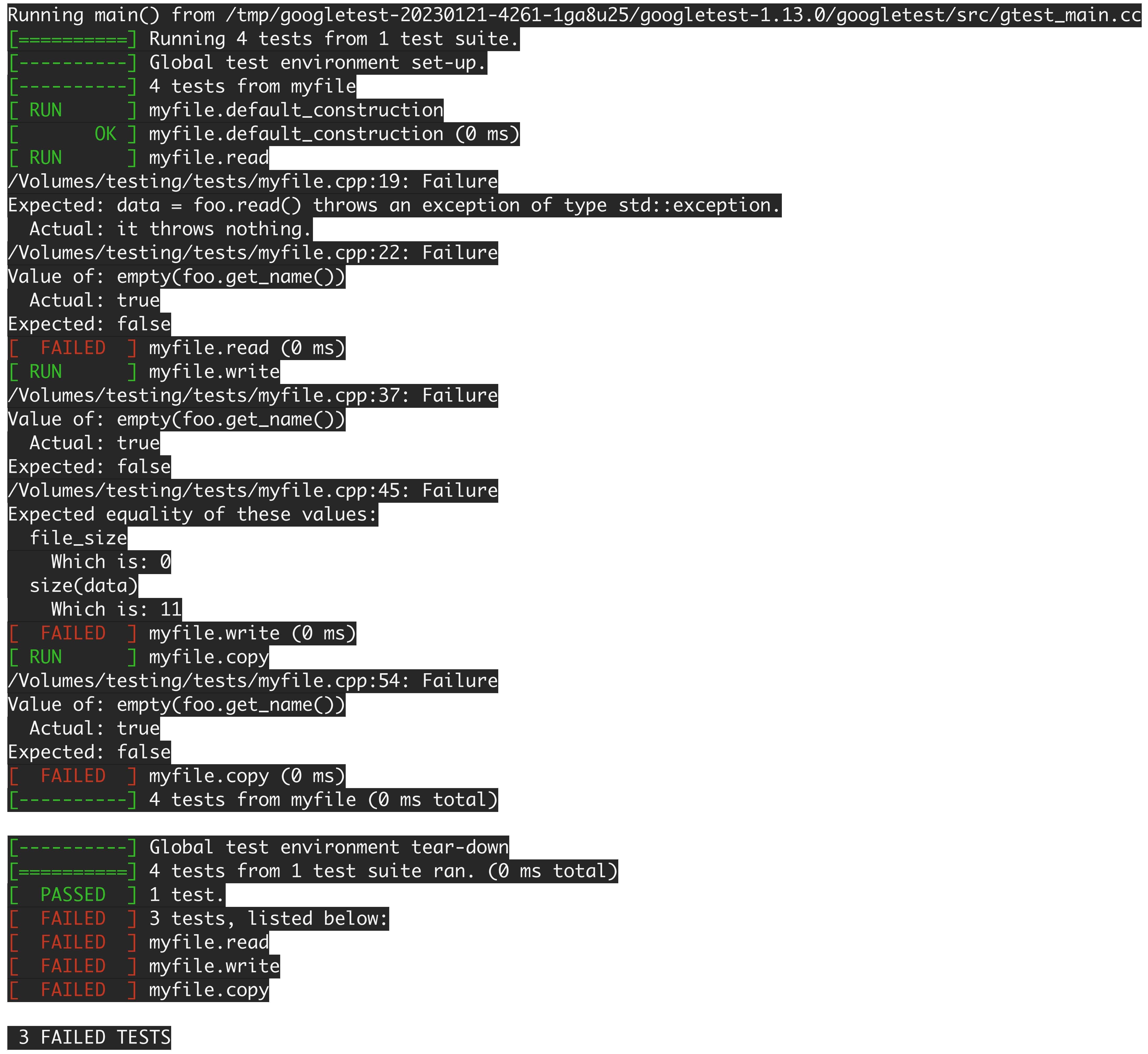
- In a “myfile.cpp” file.
- #include <gtest/gtest.h>
- #include "../library/myfile.hpp"
- TEST(myfile, copy)



```
TEST(myfile, copy)
{
    const auto data = std::string{"hello world"};
    constexpr auto file_path = "/tmp/foo-bar-roo";
    auto foo = myfile();
    EXPECT_NO_THROW(foo.open(file_path));
    EXPECT_FALSE(empty(foo.get_name()));
    EXPECT_TRUE(foo.is_open());
    auto copy = foo;
    EXPECT_TRUE(copy == foo);
    foo.close();
    EXPECT_FALSE(foo.is_open());
    EXPECT_TRUE(copy.is_open());
    EXPECT_NO_THROW(copy.write(data));
}
```

MYFILE RESULTS

- **Running…**



```
Running main() from /tmp/googletest-20230121-4261-1ga8u25/googletest-1.13.0/googletest/src/gtest_main.cc
[==========] Running 4 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 4 tests from myfile
[ RUN      ] myfile.default_construction
[       OK ] myfile.default_construction (0 ms)
[ RUN      ] myfile.read
/Volumes/testing/tests/myfile.cpp:19: Failure
Expected: data = foo.read() throws an exception of type std::exception.
  Actual: it throws nothing.
/Volumes/testing/tests/myfile.cpp:22: Failure
Value of: empty(foo.get_name())
  Actual: true
Expected: false
[  FAILED  ] myfile.read (0 ms)
[ RUN      ] myfile.write
/Volumes/testing/tests/myfile.cpp:37: Failure
Value of: empty(foo.get_name())
  Actual: true
Expected: false
/Volumes/testing/tests/myfile.cpp:45: Failure
Expected equality of these values:
  file_size
    Which is: 0
  size(data)
    Which is: 11
[  FAILED  ] myfile.write (0 ms)
[ RUN      ] myfile.copy
/Volumes/testing/tests/myfile.cpp:54: Failure
Value of: empty(foo.get_name())
  Actual: true
Expected: false
[  FAILED  ] myfile.copy (0 ms)
[----------] 4 tests from myfile (0 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 3 tests, listed below:
[  FAILED  ] myfile.read
[  FAILED  ] myfile.write
[  FAILED  ] myfile.copy

 3 FAILED TESTS
```

CATCH 2: CLIFF...