Will Your Program Still Be Correct Next Year?

Björn Fahller

2025

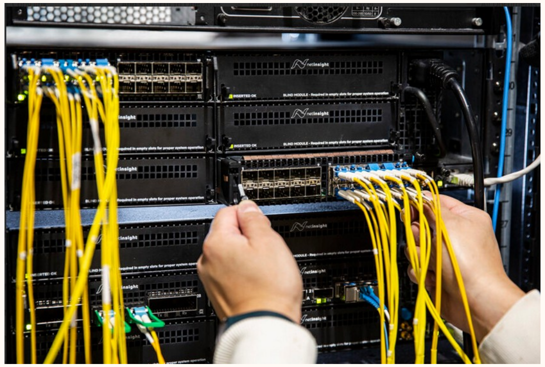
Will your program still be correct next year?

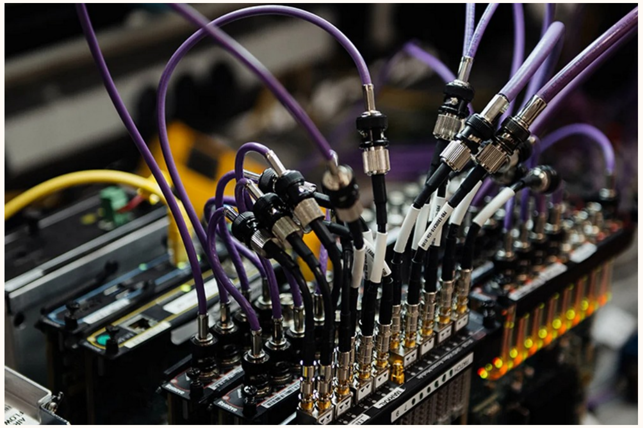
or

How to prevent evolution from taking wrong turns

Björn Fahller

netinsight





An example

```
class histogram {
public:
    void insert(const std::string& s);







    void remove(std::string_view s);



    size_t operator[](std::string_view s) const;

    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

An example

```cpp
class histogram {
public:
    void insert(const std::string& s) {
        auto [iterator, inserted] = words_.try_emplace(s, 1);
        if (!inserted) {
            ++iterator->second;
        }
    }
    void remove(std::string_view s);


    size_t operator[](std::string_view s) const;

    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

An example

```cpp
class histogram {
public:
    void insert(const std::string& s);

    void remove(std::string_view s) {
        auto iter = words_.find(s);
        assert(iter != words_.end());
        if (--iter->second == 0) {
            words_.erase(iter);
        }
    }
    size_t operator[](std::string_view s) const;

    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

An example

```
class histogram {
public:
    void insert(const std::string& s);

    void remove(std::string_view s);


    size_t operator[](std::string_view s) const {
        auto iter = words_.find(s);
        return iter == words_.end() ? 0 : iter->second;
    }


    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

An example

```
class histogram {
public:
    void insert(const std::string& s);

    void remove(std::string_view s);


    size_t operator[](std::string_view s) const;





    auto begin() const { return words_.begin();}
    auto end() const { return words_.end();}
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

An example

```
class histogram {
public:
    void insert(const std::string& s);








    void remove(std::string_view s);



    size_t operator[](std::string_view s) const;

    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

An example

```
class histogram {
public:
    void insert(const std::strin

    void remove(std::string_view

    size_t operator[](std::strin

    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

```
int main() {
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    std::println("{}", h);
    h.remove("baz");
    h.remove("foo");
    std::println("{}", h);
    std::println("{}", h["foo"]);
    std::println("{}", h["baz"]);
}
```

An example

```
class histogram {
public:
    void insert(const std::strin



    void remove(std::string_view


    size_t operator[](std::strin

    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size
};
```

```
int main() {
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    std::println("{}", h);
    h.remove("baz");
    h.remove("foo");
    std::println("{}", h);
    std::println("{}", h["foo"]);
    std::println("{}", h["baz"]);
}
```

```
[("bar", 1), ("baz", 1), ("foo", 2)]
[("bar", 1), ("foo", 1)]
1
0
```

An example

```
class histogram {
public:
    void insert(const std::string

    void remove(std::string_view

    size_t operator[](std::string

    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size
};
```

```
int main() {
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    std::println("{}", h);
    h.remove("baz");
    h.remove("foo");
    std::println("{}", h);
    std::println("{}", h["foo"]);
    std::println("{}", h["baz"]);
}
```

```
[("bar", 1), ("baz", 1), ("foo", 2)]
[("bar", 1), ("foo", 1)]
1
0
```

An example

```
class histogram {
public:
    void insert(const std::strin




    void remove(std::string_view


    size_t operator[](std::strin

    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size
};
```

```
int main() {
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    std::println("{}", h);
    h.remove("baz");
    h.remove("foo");
    std::println("{}", h);
    std::println("{}", h["foo"]);
    std::println("{}", h["baz"]);
}
```

```
[("bar", 1), ("baz", 1), ("foo", 2)]
[("bar", 1), ("foo", 1)]
1
0
```

An example

```
class histogram {
public:
    void insert(const std::string
    
    
    
    
    
    void remove(std::string_view
    
    
    size_t operator[](std::string
    
    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size
};
```

```
int main() {
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    std::println("{}", h);
    h.remove("baz");
    h.remove("foo");
    std::println("{}", h);
    std::println("{}", h["foo"]);
    std::println("{}", h["baz"]);
}
```

```
[("bar", 1), ("baz", 1), ("foo", 2)]
[("bar", 1), ("foo", 1)]
1
0
```

An example

```
class histogram {
public:
    void insert(const std::strin

    void remove(std::string_view

    size_t operator[](std::strin

    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size
};
```

```
int main() {
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    std::println("{}", h);
    h.remove("baz");
    h.remove("foo");
    std::println("{}", h);
    std::println("{}", h["foo"]);
    std::println("{}", h["baz"]);
}
```

```
[("bar", 1), ("baz", 1), ("foo", 2)]
[("bar", 1), ("foo", 1)]
1
0
```

An example

```
class histogram {
public:
    void insert(const std::strin
```

```
    void remove(std::string_view
```

```
    size_t operator[](std::strin

    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size
};
```

```
int main() {
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    std::println("{}", h);
    h.remove("baz");
    h.remove("foo");
    std::println("{}", h);
    std::println("{}", h["foo"]);
    std::println("{}", h["baz"]);
}
```

```
[("bar", 1), ("baz", 1), ("foo", 2)]
[("bar", 1), ("foo", 1)]
1
0
```

An example

```
class histogram {
public:
    void insert(const std::strin

    void remove(std::string_view

    size_t operator[](std::strin

    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size
};
```

```
int main() {
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    std::println("{}", h);
    h.remove("baz");
    h.remove("foo");
    std::println("{}", h);
    std::println("{}", h["foo"]);
    std::println("{}", h["baz"]);
}
```

```
[("bar", 1), ("baz", 1), ("foo", 2)]
[("bar", 1), ("foo", 1)]
1
0
```

An example

```
class histogram {
public:
    void insert(const std::strin



    void remove(std::string_view


    size_t operator[](std::strin

    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size
};
```

```
int main() {
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    std::println("{}", h);
    h.remove("baz");
    h.remove("foo");
    std::println("{}", h);
    std::println("{}", h["foo"]);
    std::println("{}", h["baz"]);
}
```

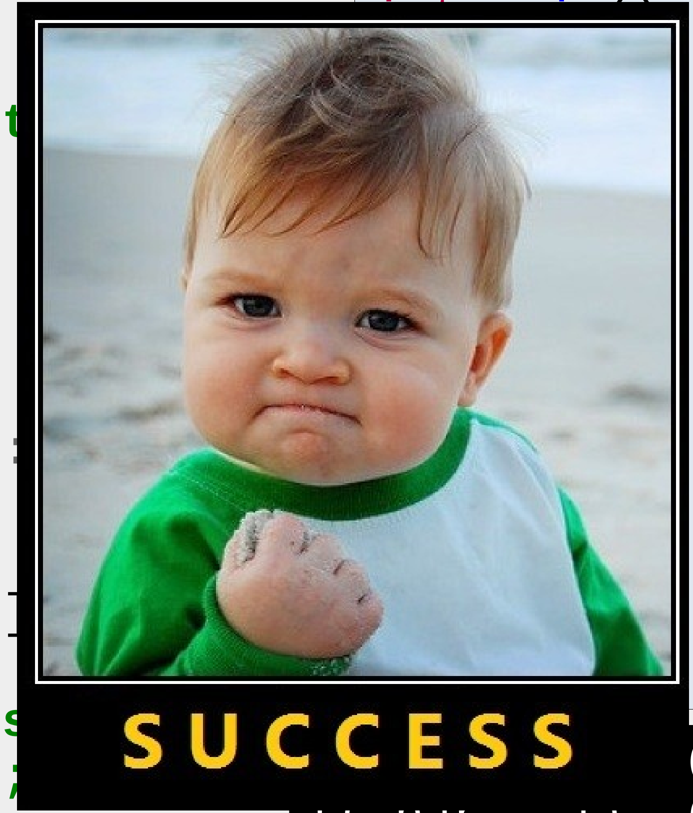
```
[("bar", 1), ("baz", 1), ("foo", 2)]
[("bar", 1), ("foo", 1)]
1
0
```

An example

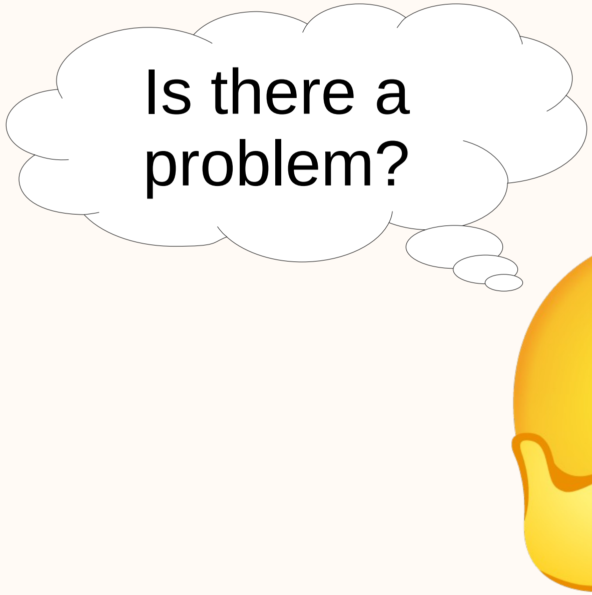
```
class histogram {
public:
    void insert(const



    void remove(std::


    size_t operator[]

    auto begin() cons
    auto end() const;
private:
    std::map<std::string, size
};
```

```
{
am h;
t("foo");
t("bar");
t("foo");
t("baz");
intln("{}", h);
e("baz");
e("foo");
intln("{}", h);
intln("{}", h["foo"]);
intln("{}", h["baz"]);
```

```
("baz", 1), ("foo", 2)]
[("bar", 1), ("foo", 1)]
1
0
```

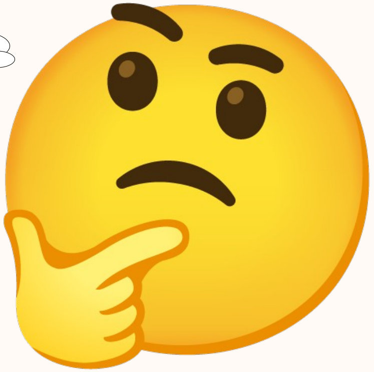


Or…?



Or…?

Is there a problem?



No one will know if a future change introduces a bug

An example

```
class histogram {
public:
    void insert(const std::string& s);






    void remove(std::string_view s);


    size_t operator[](std::string_view s) const;

    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

An example

```
class histogram {
public:
    void insert(con

    void remove(std

    size_t operator

    auto begin() co
    auto end() cons
private:
    std::map<std::s
};
```

```
int main() {
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    std::pair<std::string_view, size_t> expected1[] {
        {"bar", 1}, {"baz", 1}, {"foo", 2}
    };
    assert(std::ranges::equal(h, expected1));
    h.remove("baz");
    h.remove("foo");
    std::pair<std::string_view, size_t> expected2[] {
        {"bar", 1}, {"foo", 1}
    };
    assert(std::ranges::equal(h, expected2));
    assert(h["foo"] == 1);
    assert(h["baz"] == 0);
}
```

An example

```
class histogram {
public:
    void insert(con

    void remove(std

    size_t operator

    auto begin() co
    auto end() cons
private:
    std::map<std::s
};
```

```
int main() {
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    std::pair<std::string_view, size_t> expected1[] {
        {"bar", 1}, {"baz", 1}, {"foo", 2}
    };
    assert(std::ranges::equal(h, expected1));
    h.remove("baz");
    h.remove("foo");
    std::pair<std::string_view, size_t> expected2[] {
        {"bar", 1}, {"foo", 1}
    };
    assert(std::ranges::equal(h, expected2));
    assert(h["foo"] == 1);
    assert(h["baz"] == 0);
}
```

An example

```
class histogram {
public:
    void insert(con

    void remove(std

    size_t operator

    auto begin() co
    auto end() cons
private:
    std::map<std::s
};
```

```
int main() {
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    std::pair<std::string_view, size_t> expected1[] {
        {"bar", 1}, {"baz", 1}, {"foo", 2}
    };
    assert(std::ranges::equal(h, expected1));
    h.remove("baz");
    h.remove("foo");
    std::pair<std::string_view, size_t> expected2[] {
        {"bar", 1}, {"foo", 1}
    };
    assert(std::ranges::equal(h, expected2));
    assert(h["foo"] == 1);
    assert(h["baz"] == 0);
}
```

An example

```
class histogram {
public:
    void insert(con



    void remove(std


    size_t operator

    auto begin() co
    auto end() cons
private:
    std::map<std::s
};
```

```
int main() {
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    std::pair<std::string_view, size_t> expected1[] {
        {"bar", 1}, {"baz", 1}, {"foo", 2}
    };
    assert(std::ranges::equal(h, expected1));
    h.remove("baz");
    h.remove("foo");
    std::pair<std::string_view, size_t> expected2[] {
        {"bar", 1}, {"foo", 1}
    };
    assert(std::ranges::equal(h, expected2));
    assert(h[
    assert(h[
}
```

```
> ./test_histogram && print "PASS"
PASS
>
```

An example

```
class histogram {

    std::map<std::s
};
```

```
int main() {
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    std::pair<std::string_view, size_t> expected1[] {
        {"bar", 1}, {"baz", 1}, {"foo", 2}
    };
    assert(std::ranges::equal(h, expected1));
    h.remove("baz");
    h.remove("foo");
    std::pair<std::string_view, size_t> expected2[] {
        {"bar", 1}, {"foo", 1}
    };
    assert(std::ranges::equal(h, expected2));
    assert(h[
    assert(h[
}
```

```
> ./test_histogram && print "PASS"
PASS
>
```

Some types of tests

feature tests

unit tests

ad-hoc tests

acceptance tests

smoke tests

network tests

performance tests

stability tests

stress tests

integration tests

production tests

fuzz tests

Some types of tests

Type	Purpose
Fuzz tests	Find bugs
Ad-hoc tests	Find bugs
Acceptance test	Verify compliance with requirements
Production tests	Find HW problems
Unit tests	Show that the code does what’s intended
Performance tests	Ensure performance meets requirements (or doesn’t degrade)
Stability tests	Find state degradation
Integration tests	Find misunderstandings in interfaces
Stress tests	Verify that priorities are right
Smoke tests	Stop wasted time

Some types of tests

Type	Receiver of result
Fuzz tests	Developer
Ad-hoc tests	Team(s)
Acceptance test	Team(s) / product lead / client
Production tests	Factory
Unit tests	Developer
Performance tests	Team / product lead
Stability tests	Team
Integration tests	Team(s)
Stress tests	Team(s)
Smoke tests	Team(s)

Some types of tests

Type	Desired outcome of failures
Fuzz tests	Discovered missed cases
Ad-hoc tests	Discovered missed functionality or mis-modeled couplings
Acceptance test	Identify misunderstood or missed requirements
Production tests	Prevent shipping of defective products
Unit tests	Understand what is broken
Performance tests	Identify bottle necks
Stability tests	Find memory leaks, cumulative errors
Integration tests	Identify misunderstanding between teams/developers
Stress tests	Find modeling errors regarding priorities
Smoke tests	Avoid wasting time with expensive tests on a broken build

Some types of tests

Type	Desired outcome of failures
Fuzz tests	Discovered missed cases
Ad-hoc tests	Discovered missed functionality or mis-modeled couplings
Acceptance test	Identify misunderstood or missed requirements
Production tests	Prevent shipping of defective products
Unit tests	Understand what is broken
Performance tests	Identify bottle necks
Stability tests	Find memory leaks, cumulative errors
Integration tests	Identify misunderstanding between teams/developers
Stress tests	Find modeling errors regarding priorities
Smoke tests	Avoid wasting time with expensive tests on a broken build

Some types of tests

Type	Desired outcome of failures
Fuzz tests	Discovered missed cases
Ad-hoc tests	Discovered missed functionality or mis-modeled couplings
Acceptance test	Identify misunderstood or missed requirements
Production tests	Prevent shipping of defective products
Unit tests	Understand what is broken
Performance tests	Identify bottle necks
Stability tests	Find memory leaks, cumulative errors
Integration tests	Identify misunderstanding between teams/developers
Stress tests	Find modeling errors regarding priorities
Smoke tests	Avoid wasting time with expensive tests on a broken build

Some types of tests

Type	Desired outcome of failures
Fuzz tests	Discovered missed cases
Ad-hoc tests	Discovered missed functionality or mis-modeled couplings
Acceptance test	Identify misunderstood or missed requirements
Production tests	Prevent shipping of defective products
Unit tests	Understand what is broken
Performance tests	Identify bottle necks
Stability tests	Find memory leaks, cumulative errors
Integration tests	Identify misunderstanding between teams/developers
Stress tests	Find modeling errors regarding priorities
Smoke tests	Avoid wasting time with expensive tests on a broken build

Some types of tests

Type	Desired outcome of failures
Fuzz tests	Discovered missed cases
Ad-hoc tests	Disc[illegible]sed functionality or mis-modeled couplings
Acceptance test	[illegible] or missed requirements
Production tests	[illegible]ctive products
Unit tests	[illegible]en
Perfor[illegible]ce test[illegible]	[illegible]
Stabilit[illegible]ests	[illegible], cumulative errors
Integration test[illegible]	Identify misunderstanding between teams/developers
S[illegible]sts	Find modeling errors regarding priorities
[illegible]sts	Avoid wasting time with expensive tests on a broken build

These tests say what is not working as expected

Some types of tests

Type	Desired outcome of failures
Fuzz tests	Discovered missed cases
Ad-hoc tests	Disc [illegible] sed functionalit [illegible] led couplings
Acceptance test	[illegible]
Production tests	[illegible]
Unit tests	[illegible] en
Perfor[illegible]ce test	[illegible]
Stabilit[illegible]ests	[illegible], cumulative errors
Integration test	Identify misunderstanding between teams/develop
S[illegible]sts	Find modeling errors regarding priorities
[illegible]sts	Avoid wasting time with expensive tests on a b

These tests say what is not working as expected

So they better say what the expectations are

Some problems

```
class histogram {
public:
    void insert(con

    void remove(std

    size_t operator

    auto begin() co
    auto end() cons
private:
    std::map<std::s
};
```

```
int main() {
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    std::pair<std::string_view, size_t> expected1[] {
        {"bar", 1}, {"baz", 1}, {"foo", 2}
    };
    assert(std::ranges::equal(h, expected1));
    h.remove("baz");
    h.remove("foo");
    std::pair<std::string_view, size_t> expected2[] {
        {"bar", 1}, {"foo", 1}
    };
    assert(std::ranges::equal(h, expected2));
    assert(h["foo"] == 1);
    assert(h["baz"] == 0);
}
```

Some problems

```
class histogram {
public:
    void insert(con



                 td


    size_t oper    or

    auto begin() co
    auto end() cons
private:
    std::map<std::s
};
```

```
int main() {
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    std::pair<std::string_view, size_t> expected1[] {
        {"bar", 1}, {"baz", 1}, {"foo", 2}
    };
    assert(std::ranges::equal(h, expected1));
    h.remove("baz");
    h.remove("foo");
    std::pair<std::string_view, size_t> expected2[] {
        {"bar", 1}, {"foo", 1}
    };
    assert(std::ranges::equal(h, expected2));
    assert(h["foo"] == 1);
    assert(h["baz"] == 0);
}
```

Failed asserts leads to detective work to understand what the problem is.

Some problems

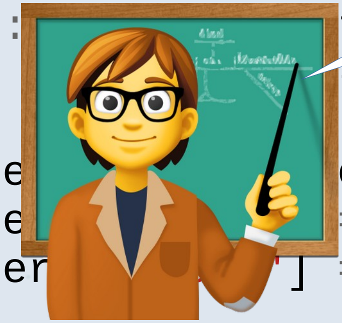
```
class histogram {
public:
    void insert(con



                    td



    size_t oper   or

    auto begin() co
    auto end() cons
private:
    std::map<std::s
};
```

```
int main() {
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    std::pair<std::strin
        {"bar", 1}, {"ba
    };
    assert(std::ranges::
    h.remove("baz");
    h.remove("foo");
    std:          tri  g_view, size_t> expected2[] {
                  {"foo", 1}
    };
    asse          es::equal(h, expected2));
    asse             == 1);
    asser          '] == 0);
}
```

Failed asserts leads to detective work to understand what the problem is.

Write your tests as a set of requirements, each stating a functionality. A failed test will always tell you **what** doesn't work.

Let the failure message explain how the problem was found.



An example

```cpp
int main() {
    {
        fputs("A default constructed histogram has no words",
              stderr);
        histogram h;
        assert(h.begin() == h.end());
        fputs(" PASS!\n", stderr);
    }
    {
        fputs("When a word is inserted,"
              " it has a ref count of one with operator[]",
              stderr);
        histogram h;
        h.insert("foo");
        assert(h["foo"] == 1);
        fputs(" PASS!\n", stderr);
    }
}
```

An example

```
int main() {
    {
```

```
A default constructed histogram has no words PASS!
When a word is inserted, it has a ref count of one with operator[]
histogram_test: histogram_test.cpp:16: int main():
Assertion `h["foo"] == 1' failed.
Program terminated with signal: SIGABRT
```

```
    }
    {
        fputs("When a word is inserted,"
              " it has a ref count of one with operator[]",
              stderr);
        histogram h;
        h.insert("foo");
        assert(h["foo"] == 1);
        fputs(" PASS!\n", stderr);
    }
}
```

An example

```
int main() {
    {
```

```
A default constructed histogram has no words PASS!
When a word is inserted, it has a ref count of one with operator[]
histogram_test: histogram_test.cpp:16: int main():
Assertion `h["foo"] == 1' failed.
Program terminated with signal: SIGABRT
```

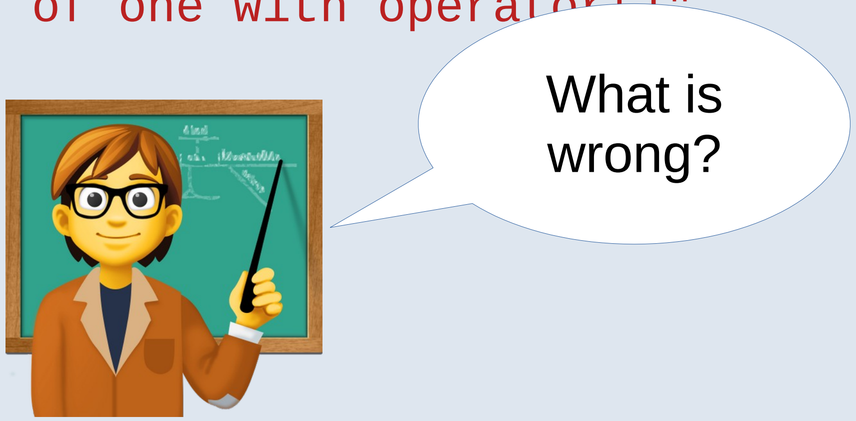
```
    }
    {
        fputs("When a word is inserted,"
              " it has a ref count of one with operator[]"
              stderr);
        histogram h;
        h.insert("foo");
        assert(h["foo"] == 1);
        fputs(" PASS!\n", stderr);
    }
}
```

An example

```
int main() {
    {
```

```
A default constructed histogram has no words PASS!
When a word is inserted, it has a ref count of one with operator[]
histogram_test: histogram_test.cpp:16: int main():
Assertion `h["foo"] == 1' failed.
Program terminated with signal: SIGABRT
```

```
    }
    {
        fputs("When a word is inserted,"
              " it has a ref count of one with operator[]"
              stderr);
        histogram h;
        h.insert("foo");
        assert(h["foo"] == 1);
        fputs(" PASS!\n", stderr);
    }
}
```



An example

```
int main() {
    {
```

```
A default constructed histogram has no words PASS!
When a word is inserted, it has a ref count of one with operator[]
histogram_test: histogram_test.cpp:16: int main():
Assertion `h["foo"] == 1' failed.
Program terminated with signal: SIGABRT
```

```
    }
    {
        fputs("When a word is inserted,"
              " it has a ref count of one with oper
              stderr);
        histogram h;
        h.insert("foo");
        assert(h["foo"] == 1);
        fputs(" PASS!\n", stderr);
    }
}
```

An example

```
int main() {
    {
```

```
A default constructed histogram has no words PASS!
When a word is inserted, it has a ref count of one with operator[]
histogram_test: histogram_test.cpp:16: int main():
Assertion `h["foo"] == 1' failed.
Program terminated with signal: SIGABRT
```

```
    }
    {
        fputs("When a word is inserted,"
              " it has a ref count of one with oper
              stderr);
        histogram h;
        h.insert("foo");
        assert(h["foo"] == 1);
        fputs(" PASS!\n", stderr);
    }
}
```

How was the problem found?

Use a test framework

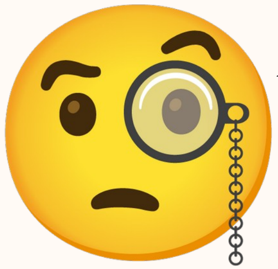
Some FOSS frameworks

Name	Where	License
doctest	https://github.com/doctest/doctest	MIT
catch2	https://github.com/catchorg/catch2	BSL-1.0
gtest	https://github.com/google/googletest	BSD 3-clause
boost test	https://github.com/boostorg/test	BSL-1.0
criterion	https://github.com/Snaipe/Criterion	MIT

Use a test framework

Some FOSS frameworks

Name	Where	License
doctest	https://github.com/doctest/doctest	MIT
catch2	https://github.com/catchorg/catch2	BSL-1.0
gtest	https://github.com/google/googletest	BSD 3-clause
boost test	https://github.com/boostorg/test	BSL-1.0
criterion	https://github.com/Snaipe/Criterion	MIT



There are also commercial tools

Example using doctest

```cpp
#include <histogram.h>

#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include <doctest.h>

TEST_CASE("A default constructed histogram has no words")
{
    histogram h;
    REQUIRE(h.begin() == h.end());
}

TEST_CASE("When a word is inserted it has a ref count of 1 with []")
{
    histogram h;
    h.insert("foo");
    REQUIRE(h["foo"] == 1);
}
```

Example using doctest

```
#include <histogram.h>

#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include <doctest.h>

TEST_CASE("A default constructe                        rds")
{
    histogram h;
    REQUIRE(h.begin() == h.end());
}

TEST_CASE("When a word is inserted it has a ref count of 1 with []")
{
    histogram h;
    h.insert("foo");
    REQUIRE(h["foo"] == 1);
}
```

#include the header of the code to test first.

Example using doctest

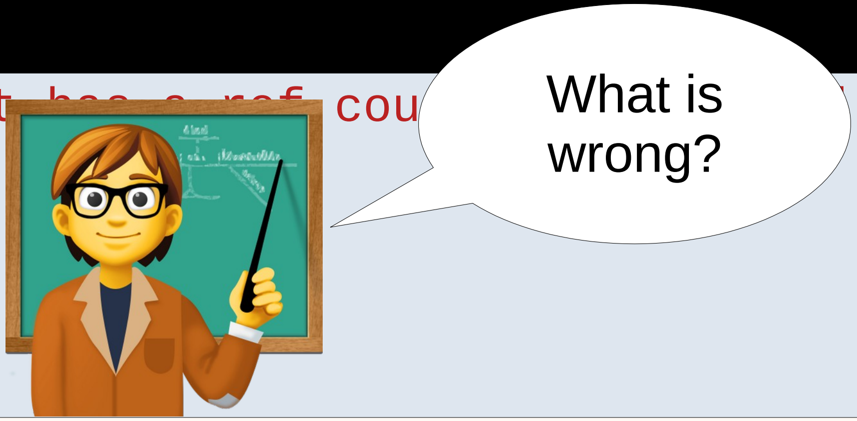
```
===============================================================================
htest.cpp:13:
TEST CASE:  When a word is inserted, it has a ref count of 1 with []
htest.cpp:17: FATAL ERROR: REQUIRE( h["foo"] == 1 ) is NOT correct!
Values:
 REQUIRE( 0 == 1 )
===============================================================================
[doctest] test cases: 2 | 1 passed | 1 failed | 0 skipped
[doctest] assertions: 2 | 1 passed | 1 failed |
[doctest] Status: FAILURE!
```

```
TEST_CASE("When a word is inserted it has a ref count of 1 with []")
{
    histogram h;
    h.insert("foo");
    REQUIRE(h["foo"] == 1);
}
```

Example using doctest

```
================================================================
htest.cpp:13:
TEST CASE:  When a word is inserted, it has a ref count of 1 with []
htest.cpp:17: FATAL ERROR: REQUIRE( h["foo"] == 1 ) is NOT correct!
Values:
 REQUIRE( 0 == 1 )
================================================================
[doctest] test cases: 2 | 1 passed | 1 failed | 0 skipped
[doctest] assertions: 2 | 1 passed | 1 failed |
[doctest] Status: FAILURE!
```

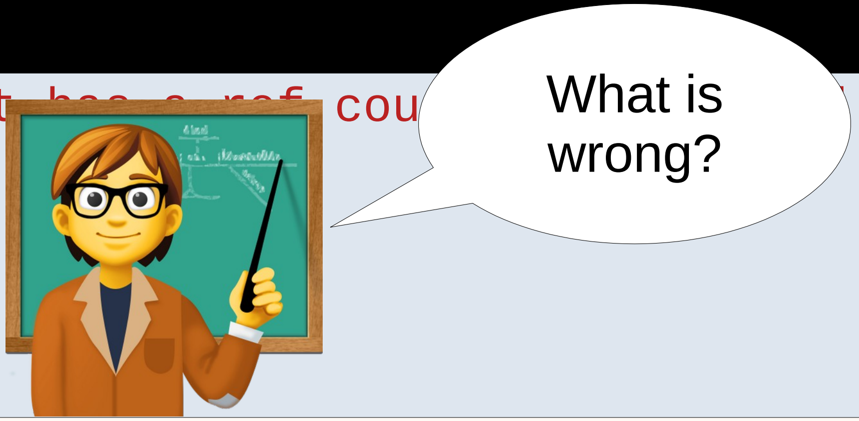
```
TEST_CASE("When a word is inserted it has a ref cou              ")
{
    histogram h;
    h.insert("foo");
    REQUIRE(h["foo"] == 1);
}
```



Example using doctest

```
===============================================================================
htest.cpp:13:
TEST CASE:  When a word is inserted, it has a ref count of 1 with []
htest.cpp:17: FATAL ERROR: REQUIRE( h["foo"] == 1 ) is NOT correct!
Values:
 REQUIRE( 0 == 1 )
===============================================================================
[doctest] test cases: 2 | 1 passed | 1 failed | 0 skipped
[doctest] assertions: 2 | 1 passed | 1 failed |
[doctest] Status: FAILURE!
```

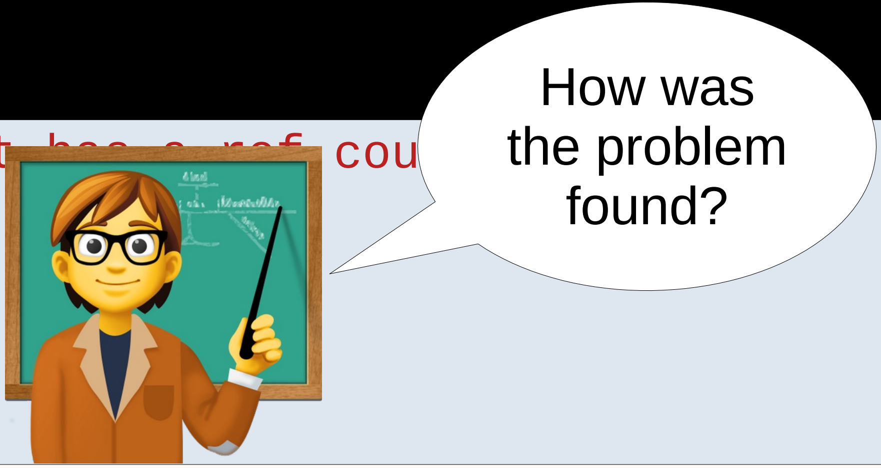
```
TEST_CASE("When a word is inserted it has a ref cou                 ")
{
    histogram h;
    h.insert("foo");
    REQUIRE(h["foo"] == 1);
}
```



Example using doctest

```
===================================================================
htest.cpp:13:
TEST CASE:  When a word is inserted, it has a ref count of 1 with []
htest.cpp:17: FATAL ERROR: REQUIRE( h["foo"] == 1 ) is NOT correct!
Values:
 REQUIRE( 0 == 1 )
===================================================================
[doctest] test cases: 2 | 1 passed | 1 failed | 0 skipped
[doctest] assertions: 2 | 1 passed | 1 failed |
[doctest] Status: FAILURE!
```

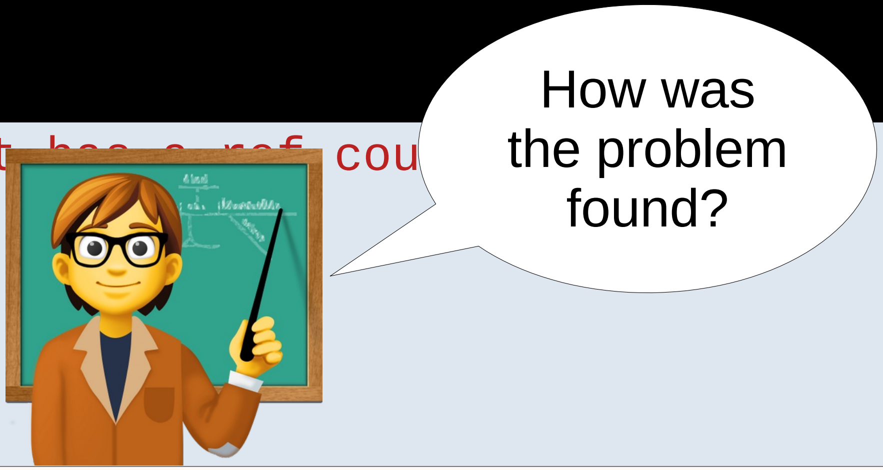
```
TEST_CASE("When a word is inserted it                      cou              )
{
    histogram h;
    h.insert("foo");
    REQUIRE(h["foo"] == 1);
}
```



Example using doctest

```
===============================================================================
htest.cpp:13:
TEST CASE:  When a word is inserted, it has a ref count of 1 with []
htest.cpp:17: FATAL ERROR: REQUIRE( h["foo"] == 1 ) is NOT correct!
Values:
 REQUIRE( 0 == 1 )
===============================================================================
[doctest] test cases: 2 | 1 passed | 1 failed | 0 skipped
[doctest] assertions: 2 | 1 passed | 1 failed |
[doctest] Status: FAILURE!
```

```
TEST_CASE("When a word is inserted it has a ref cou                   )
{
    histogram h;
    h.insert("foo");
    REQUIRE(h["foo"] == 1);
}
```



More functions

```
class histogram {
public:
    void insert(const std::string& s);

    void remove(std::string_view s);

    size_t operator[](std::string_view s) const;
```

More functions

```
class histogram {
public:
    void insert(const std::string& s);

    void remove(std::string_view s);

    size_t operator[](std::string_view s) co
```

We need a way to query the number of words in the histogram

More functions

```
class histogram {
public:
    void insert(const std::string& s);

    void remove(std::string_view s);

    size_t operator[](std::string_view s) const;

    [[nodiscard]] bool empty() const { return words_.empty(); }
    [[nodiscard]] size_t size() const { return words_.size(); }
    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

More tests

```
#include <histogram.h>

#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include <doctest.h>

TEST_CASE("A default constructed histogram has no words")
{
    histogram h;
    REQUIRE(h.begin() == h.end());
    REQUIRE(h.empty());
    REQUIRE(h.size() == 0);
}
```

More tests

```
#include <histogram.h>

#define DOCTEST_CONFIG_IMPLEMENT_WITH_
#include <doctest.h>

TEST_CASE("A default constructed histogr    as no words")
{
    histogram h;
    REQUIRE(h.begin() == h.end());
    REQUIRE(h.empty());
    REQUIRE(h.size() == 0);
}
```

This just made a lot of people cringe!

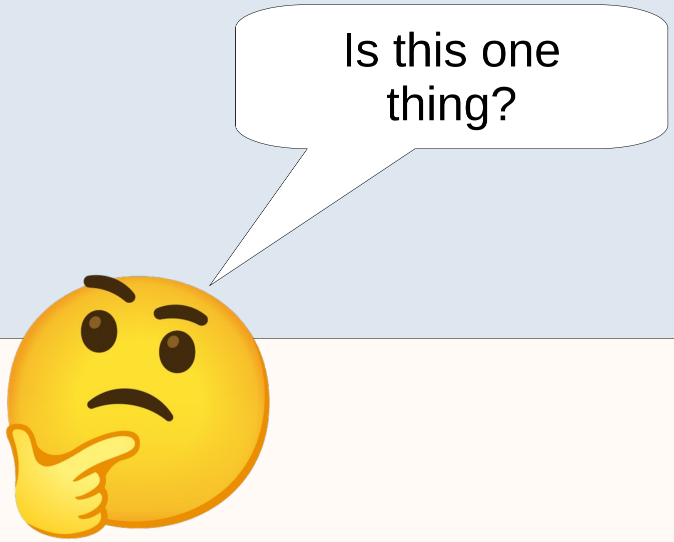
More tests

```
#include <histogram.h>

#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include <doctest.h>

TEST_CASE("A default constructed histogram has r
{
    histogram h;
    REQUIRE(h.begin() == h.end());
    REQUIRE(h.empty());
    REQUIRE(h.size() == 0);
}
```

A test case should test only one thing

More tests

```
#include <histogram.h>

#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include <doctest.h>

TEST_CASE("A default constructed histogram has no words")
{
    histogram h;
    REQUIRE(h.begin() == h.end());
    REQUIRE(h.empty());
    REQUIRE(h.size() == 0);
}
```



More tests

```
TEST_CASE("adding words affects size")
{
    GIVEN("a histogram with two unique words") {
        histogram h;
        h.insert("foo");
        h.insert("bar");
        THEN("the size is two") {
            REQUIRE(h.size() == 2);
        }
        AND_WHEN("adding a word that already exists") {
            h.insert("foo");
            THEN("the size is still two") {
                REQUIRE(h.size() == 2);
            }
        }
        AND_WHEN("adding a new word") { ...
```

More tests

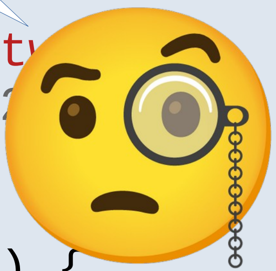
```
TEST_CASE("adding words affects size")
{
    GIVEN("a histogram with two unique words")
        histogram h;
        h.insert("foo");
        h.insert("bar");
        THEN("the size is two") {
            REQUIRE(h.size() == 2);
        }
        AND_WHEN("adding a word that already      ists") {
            h.insert("foo");
            THEN("the size              wo") {
                REQUIRE(h.s          );
            }
        }
        AND_WHEN("adding a            ) { ...
```

BDD style
  GIVEN…
  WHEN…
  THEN…
Can be a powerful way of expressing tests with a common theme and a common start state

More tests

Beware the temptation to add
AND_WHEN…
    THEN…
    AND_THEN…
AND_WHEN…
    THEN…
    AND_THEN…
    AND_THEN…

```
TEST_C                    affects size")
{
    GI                    th two unique words") {


                               o") {
                               = 2);

        AND_WHEN( adding a word    at already exists") {
            h.insert("foo");
            THEN("the size is still tw
                REQUIRE(h.size() == 2
            }
        }
        AND_WHEN("adding a new word") { ...
```



More tests

```
htest.cpp:48:
TEST CASE:  adding words affects size
     Given: a histogram with two unique words
  And when: adding a new word
      Then: the size becomes three
htest.cpp:66: FATAL ERROR: REQUIRE( h.size() == 3 ) is NOT correct!
  values: REQUIRE( 2 == 3 )
====================================================================
[doctest] test cases: 4 | 3 passed | 1 failed | 0 skipped
[doctest] assertions: 9 | 8 passed | 1 failed |
[doctest] Status: FAILURE!
```

```
            THEN("the size is still two") {
                REQUIRE(h.size() == 2);
            }
        }
        AND_WHEN("adding a new word") { ...
```

More tests

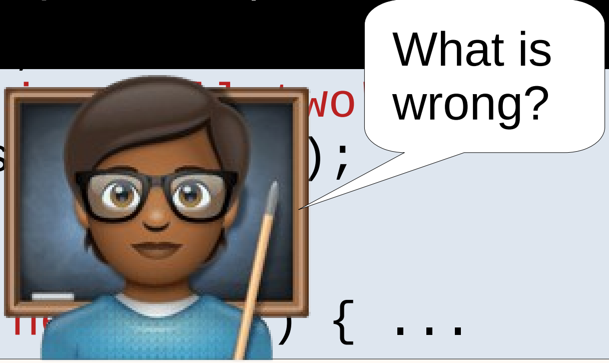
```
htest.cpp:48:
TEST CASE:  adding words affects size
     Given: a histogram with two unique words
   And when: adding a new word
      Then: the size becomes three
htest.cpp:66: FATAL ERROR: REQUIRE( h.size() == 3 ) is NOT correct!
  values: REQUIRE( 2 == 3 )
====================================================================
[doctest] test cases: 4 | 3 passed | 1 failed | 0 skipped
[doctest] assertions: 9 | 8 passed | 1 failed |
[doctest] Status: FAILURE!
```

```
            THEN("the size          wo
                REQUIRE(h.s         );
            }
        }
        AND_WHEN("adding a          ) { ...
```

More tests

```
htest.cpp:48:
TEST CASE:  adding words affects size
     Given: a histogram with two unique words
  And when: adding a new word
      Then: the size becomes three
htest.cpp:66: FATAL ERROR: REQUIRE( h.size() == 3 ) is NOT correct!
  values: REQUIRE( 2 == 3 )
====================================================================
[doctest] test cases: 4 | 3 passed | 1 failed | 0 skipped
[doctest] assertions: 9 | 8 passed | 1 failed |
[doctest] Status: FAILURE!
```

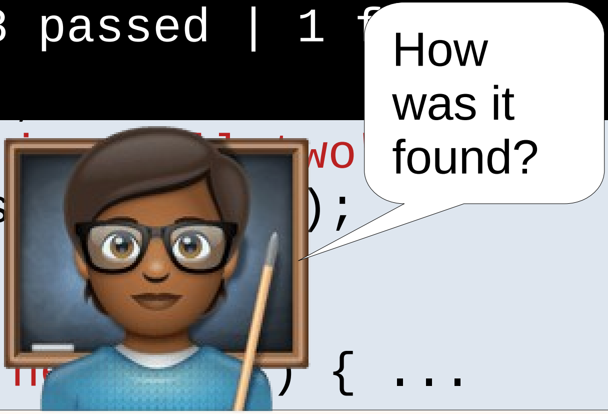
```
            THEN("the size          wo
                REQUIRE(h.s          );
            }
        }
        AND_WHEN("adding a           ) { ...
```



More tests

```
htest.cpp:48:
TEST CASE:  adding words affects size
     Given: a histogram with two unique words
  And when: adding a new word
      Then: the size becomes three
htest.cpp:66: FATAL ERROR: REQUIRE( h.size() == 3 ) is NOT correct!
  values: REQUIRE( 2 == 3 )
====================================================================
[doctest] test cases: 4 | 3 passed | 1 failed | 0 skipped
[doctest] assertions: 9 | 8 passed | 1 f
[doctest] Status: FAILURE!
```

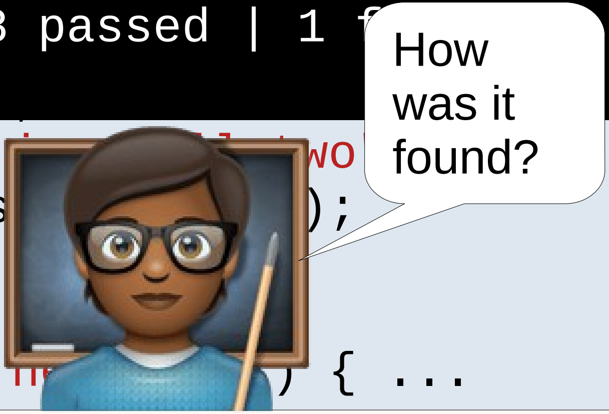
```
            THEN("the size                wo
                REQUIRE(h.s               );
            }
        }
        AND_WHEN("adding a                ) { ...
```



More tests

```
htest.cpp:48:
TEST CASE:  adding words affects size
     Given: a histogram with two unique words
  And when: adding a new word
      Then: the size becomes three
htest.cpp:66: FATAL ERROR: REQUIRE( h.size() == 3 ) is NOT correct!
  values: REQUIRE( 2 == 3 )
===================================================================
[doctest] test cases: 4 | 3 passed | 1 failed | 0 skipped
[doctest] assertions: 9 | 8 passed | 1 f
[doctest] Status: FAILURE!
```

```
            THEN("the size                 wo
                REQUIRE(h.s                  );
            }
        }
        AND_WHEN("adding a ne              ) { ...
```



More tests

```
TEST_CASE("A histogram is a range with all words and their ref count")
{
    histogram h;
    for (auto word : {"foo","bar","foo","baz","banana","baz"})
    {
        h.insert(word);
    }
    std::pair<std::string_view, size_t> expected[]{
        {"banana", 1}, {"bar", 1}, {"baz", 2}, {"foo", 2}
    };
    REQUIRE(std::ranges::equal(h, expected));
}
```

More tests

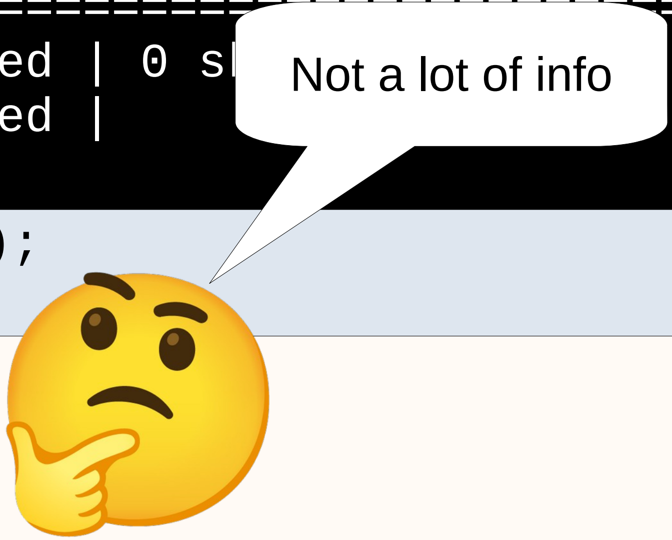
```
htest.cpp:67:
TEST CASE:  A histogram is a range with all words and their ref count
htest.cpp:77: FATAL ERROR:
 REQUIRE( std::ranges::equal(h, expected) ) is NOT correct!
  values:
 REQUIRE( false )
====================================================================
[doctest] test cases: 4 | 3 passed | 1 failed | 0 skipped
[doctest] assertions: 7 | 6 passed | 1 failed |
[doctest] Status: FAILURE!
```

```
    REQUIRE(std::ranges::equal(h, expected));
}
```

More tests

```
htest.cpp:67:
TEST CASE:  A histogram is a range with all words and their ref count
htest.cpp:77: FATAL ERROR:
 REQUIRE( std::ranges::equal(h, expected) ) is NOT correct!
  values:
 REQUIRE( false )
====================================================================
[doctest] test cases: 4 | 3 passed | 1 failed | 0 s
[doctest] assertions: 7 | 6 passed | 1 failed |
[doctest] Status: FAILURE!
```

```
    REQUIRE(std::ranges::equal(h, expected));
}
```



More tests

```cpp
#include <catch2/catch_test_macros.hpp>
#include <catch2/matchers/catch_matchers_all.hpp>

TEST_CASE("A histogram is a range with all words and their ref count")
{
    histogram h;
    for (auto word : {"foo","bar","foo","baz","banana","baz"})
    {
        h.insert(word);
    }
    std::pair<std::string_view, size_t> expected[]{
        {"banana", 1}, {"bar", 1}, {"baz", 2}, {"foo", 2}
    };
    REQUIRE_THAT(h, Catch::Matchers::RangeEquals(expected));
}
```

More tests

```
#include <catch2/catch_test_macros.hpp>
#include <catch2/matchers/catch_matchers_all.hpp>

TEST_CASE("A histogram is a range with all words and their ref count")
{
    histogram h;
    for (auto word : {"foo","bar                        ,"baz"})
    {
        h.insert(word);
    }
    std::pair<std::string_view, size_t> expected[]{
        {"banana", 1}, {"bar", 1}, {"baz", 2}, {"foo", 2}
    };
    REQUIRE_THAT(h, Catch::Matchers::RangeEquals(expected));
}
```

Another unit-test framework, Catch2

More tests

```cpp
#include <catch2/catch_test_macros.hpp>
#include <catch2/matchers/catch_matchers_all.hpp>

TEST_CASE("A histogram is a range with all words and their ref count")
{
    histogram h;
    for (auto word : {"foo","bar","foo",
    {
        h.insert(word);
    }
    std::pair<std::string_view, size_t> expected[]{
        {"banana", 1}, {"bar", 1}, {"baz", 2}, {"foo", 2}
    };
    REQUIRE_THAT(h, Catch::Matchers::RangeEquals(expected));
}
```

Compare a whole range of values

More tests

```
--------------------------------------------------------------------
A histogram is a range with all words and their ref count
--------------------------------------------------------------------
htest.cpp:75
....................................................................
htest.cpp:85: FAILED:
  REQUIRE_THAT( h, Catch::Matchers::RangeEquals(expected) )
with expansion:  { {?}, {?}, {?}, {?} }
 elements are { {?}, {?}, {?}, {?} }
=========================================================================
test cases: 4 | 3 passed | 1 failed
assertions: 7 | 6 passed | 1 failed
```

```
    };
    REQUIRE_THAT(h, Catch::Matchers::RangeEquals(expected));
}
```

More tests

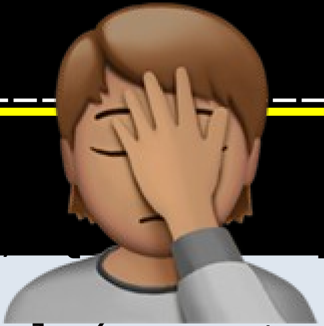
```
-------------------------------------------------------------------
A histogram is a range with all words and their ref count
-------------------------------------------------------------------
htest.cpp:75
...................................................................
htest.cpp:85: FAILED:
  REQUIRE_THAT( h, Catch::Matchers::RangeEquals(expected) )
with expansion:  { {?}, {?}, {?}, {?} }
 elements are { {?}, {?}, {?}, {?} }
===================================================================
test cases: 4 | 3 passed | 1 failed
assertions: 7 | 6 passed | 1 failed
```

```
    };
    REQUIRE_THAT(h, Catch::Matchers::RangeEquals(expected));
}
```

More tests

```
-------------------------------------------------------------------
A histogram is a range with all words and their ref count
-------------------------------------------------------------------
htest.cpp:75
...................................................................
htest.cpp:85: FAILED:
  REQUIRE_THAT( h, Catch::Matchers::RangeEquals(expected) )
with expansion:  { {?}, {?}, {?}, {?} }
 elements are { {?}, {?}, {?}, {?} }
===================================================================
test cases: 4 | 3 passed | 1 failed
assertions: 7 | 6 passed | 1 failed
```

```
    };
    REQUIRE_THAT(h, Catch::Matchers::RangeEquals(expected));
}
```



More tests

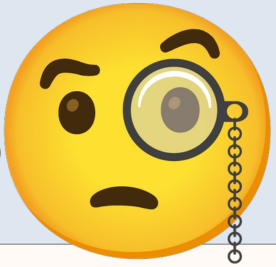
```cpp
#include <catch2/catch_test_macros.hpp>
#include <catch2/matchers/catch_matchers_all.hpp>

TEST_CASE("A histogram is a range with all words and their ref count")
{
    histogram h;
    for (auto word : {"foo","bar","foo","baz","banana","baz"})
    {
        h.insert(word);
    }
    std::pair<std::string_view, size_t> expected[]{
        {"banana", 1}, {"bar", 1}, {"baz", 2}, {"foo", 2}
    };
    REQUIRE_THAT(h, Catch::Matchers::RangeEquals(expected));
}
```

More tests

```
#include <catch2/catch_test_macros.hpp>
#include <catch2/matchers/catch_matchers_all.hpp>

TEST_CASE("A histogram is a range with all words and their ref count")
{
                                                   anana","baz"})

    }
    std::pair<std::string_view, size_t> expe    ed[]{
        {"banana", 1}, {"bar", 1}, {"baz", 2}, {"foo", 2}
    };
    REQUIRE_THAT(h, Catch::Matchers::RangeEquals(expected))
}
```

Do not fall for the temptation of writing

```
std::ostream&
operator<<(std::ostream&, std::pair...)
```



More tests

```cpp
#include <catch2/catch_test_macros.hpp>
#include <catch2/matchers/catch_matchers_all.hpp>

TEST_CASE("A histogram is a range with all words and their ref count")
{
    histogram h;
    for (auto word : {"foo","bar","foo","baz","banana","baz"})
    {
        h.insert(word);
    }
    std::pair<std::string_view, size_t> expected[]{
        {"banana", 1}, {"bar", 1}, {"baz", 2}, {"foo", 2}
    };
    REQUIRE_THAT(h, Catch::Matchers::RangeEquals(expected));
}
```

More tests

```
namespace Catch {

template<typename T, typename U>
struct StringMaker<std::pair<T,U>> {
    static std::string convert( std::pair<T,U> const& v ) {
        std::ostringstream os;
        os << "{ " << v.first << ", " << v.second << " }";
        return os.str();
    }
};

}
```

```
#i
#i

TE                                                                  )
{

    REQUIRE_THAT(h, Catch::Matchers::RangeEquals(expected));
}
```

More tests

```
#i
#i

TE                                                              )
{



    REQUIRE_THAT(h, Catch::Ma          ngeEquals(
}
```

```
namespace Catch {

template<typename T, typename U>
struct StringMaker<std::pair<T,U>> {
    static std::string convert( std::pair<T,U> const& v ) {
        std::ostringstream os;
        os << "{ " << v.first << ", " << v.second << " }";
        return os.str();
    }
};

}
```

Tell Catch2 how you want it to represent `std::pair<>` in test outputs.

More tests

```
-------------------------------------------------------------------
A histogram is a range with all words and their ref count
-------------------------------------------------------------------
htest.cpp:75
...................................................................
htest.cpp:85: FAILED:
  REQUIRE_THAT( h, Catch::Matchers::RangeEquals(expected) )
with expansion:
  { { banana, 1 }, { bar, 1 }, { baz, 1 }, { foo, 2 } }
 elements are
 { { banana, 1 }, { bar, 1 }, { baz, 2 }, { foo, 2 } }
===================================================================
test cases: 4 | 3 passed | 1 failed
assertions: 7 | 6 passed | 1 failed
```

```
}
```

Leaking implementation details

```cpp
class histogram {
public:
    void insert(const std::string& s);

    void remove(std::string_view s);

    size_t operator[](std::string_view s) const;

    [[nodiscard]] bool empty() const;
    [[nodiscard]] size_t size() const;
    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Leaking implementation details

```
class histogram {
public:
    void insert(const std::string& s);

    void remove(std::string_view s);

    size_t operator[](std::string_view s) co

    [[nodiscard]] bool empty() const;
    [[nodiscard]] size_t size() const;
    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

You need to improve performance!

Leaking implementation details

```
class histogram {
public:
    void insert(const std::string& s);

    void remove(std::string_view s);

    size_t operator[](std::string_view s) const;

    [[nodiscard]] bool empty() const;
    [[nodiscard]] size_t size() const;
    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Leaking implementation details

```
class histogram {
public:
    void insert(const std::string& s);

    void remove(std::string_view s);

    size_t operator[](std::string_view s) const;

    [[nodiscard]] bool empty() const;
    [[nodiscard]] size_t size() const;
    auto begin() const;

p

}
```

```
    struct string_hash : std::hash<std::string_view> {
        using is_transparent = void;
    };
    std::unordered_map<std::string, size_t,
                       string_hash, std::equal_to<>> words_;
```

Leaking implementation details

```
-------------------------------------------------------------------
A histogram is a range with all words and their ref count
-------------------------------------------------------------------
htest.cpp:79
..................................................................…
htest.cpp:89: FAILED:
  REQUIRE_THAT( h, Catch::Matchers::RangeEquals(expected) )
with expansion:
  { { banana, 1 }, { baz, 2 }, { bar, 1 }, { foo, 2 } }
 elements are
 { {  banana, 1 }, { bar, 1 }, { baz, 2 }, { foo, 2 } }
===================================================================
test cases: 4 | 3 passed | 1 failed
assertions: 7 | 6 passed | 1 failed
```

```
 }
     std::unordered_map<std::string, size_t,
                              string_hash, std::equal_to<>> words_;
```

Leaking implementation details

```
---------------------------------------------------------------------
A histogram is a range with all words and their ref count
---------------------------------------------------------------------
htest.cpp:79
...................................................................…
htest.cpp:89: FAILED:
  REQUIRE_THAT( h, Catch::Matchers::RangeEquals(expected) )
with expansion:
  { { banana, 1 }, { baz, 2 }, { bar, 1 }, { foo, 2 } }
 elements are
 { {  banana, 1 }, { bar, 1 }, { baz, 2 }, { foo, 2 } }
=====================================================================
test cases: 4 | 3 passed | 1 failed
assertions: 7 | 6 passed | 1 failed
```

```
}
    std::unordered_map<std::string, size_t,
                                   string_hash, std::equal_to<>> words_;
```

Leaking implementation details

```
-------------------------------------------------------------------
A histogram is a range with all words and their ref count
-------------------------------------------------------------------
htest.cpp:79
...................................................................…
htest.cpp:89: FAILED:
  REQUIRE_THAT( h, Catch::Matchers::Ranc                    ted) )
with expansion:
  { { banana, 1 }, { baz, 2 }, { bar,                       }
 elements are
 { {  banana, 1 }, { bar, 1 }, { baz, 2 },                  } }
===================================================================
test cases: 4 | 3 passed | 1 failed
assertions: 7 | 6 passed | 1 failed
```

```
}       };
    std::unordered_map<std::string, size_t,
                       string_hash, std::equal_to          _;
```

More tests

```cpp
#include <catch2/catch_test_macros.hpp>
#include <catch2/matchers/catch_matchers_all.hpp>

TEST_CASE("A histogram is a range with all words and their ref count")
{
    histogram h;
    for (auto word : {"foo","bar","foo","baz","banana","baz"})
    {
        h.insert(word);
    }
    std::pair<std::string_view, size_t> expected[]{
        {"banana", 1}, {"bar", 1}, {"baz", 2}, {"foo", 2}
    };
    REQUIRE_THAT(h, Catch::Matchers::RangeEquals(expected));
}
```

More tests

```
#include <catch2/catch_test_macros.hpp>
#include <catch2/matchers/catch_matchers_all.hpp>

TEST_CASE("A histogram is a range with all words and their ref count")
{
    histogram h;
    for (auto word : {"foo","bar","foo","baz","banana","baz"})
    {
        h.insert(word);
    }
    std::pair<std::string_view, size_t> expected[]{
        {"banana", 1}, {"bar", 1}, {"baz", 2}, {"foo", 2}
    };
    REQUIRE_THAT(h, Catch::Matchers::UnorderedRangeEquals(expected));
}
```

More tests

```
#include <catch2/catch_test_macros.hpp>
#include <catch2/matchers/catch_matchers_all.hpp>

TEST_CASE("A histogram is a range with all words and their ref count")
{
    histogram h;
    for (auto word : {"foo","bar","foo","baz","banana","baz"})
    {
        h.insert(word);
    }
    std::pair<std::string_view, size_t> expected[]{
        {"banana", 1}, {"bar", 1}, {"baz", 2}, {"foo", 2}
    };
    REQUIRE_THAT(h, Catch::Matchers::UnorderedRangeEquals(expected));
}
```

More tests

```cpp
#include <catch2/catch_test_macros.hpp>
#include <catch2/matchers/catch_matchers_all.hpp>
```

```
===============================================================================
All tests passed (7 assertions in 4 test cases)
```

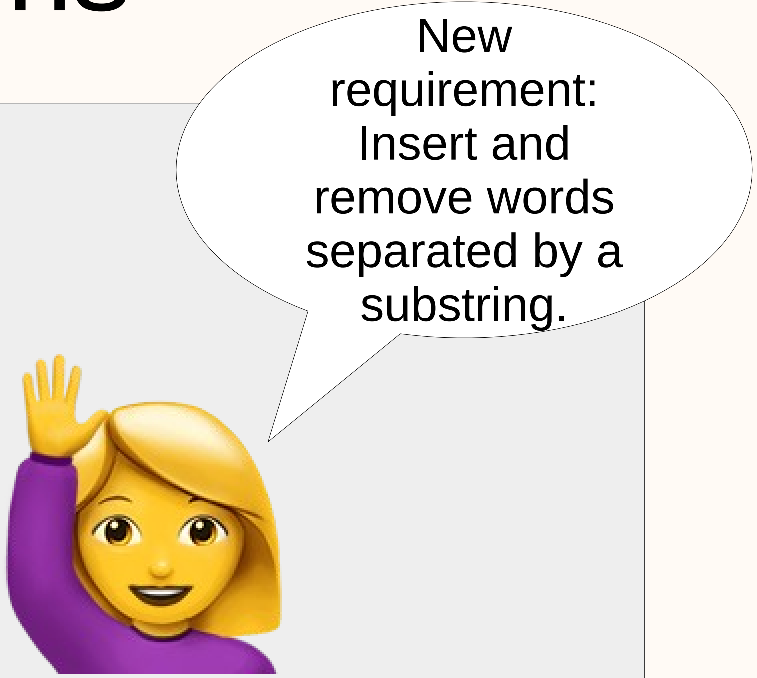
```cpp
{
    histogram h;
    for (auto word : {"foo","bar","foo","baz","banana","baz"})
    {
        h.insert(word);
    }
    std::pair<std::string_view, size_t> expected[]{
        {"banana", 1}, {"bar", 1}, {"baz", 2}, {"foo", 2}
    };
    REQUIRE_THAT(h, Catch::Matchers::UnorderedRangeEquals(expected));
}
```

Private functions

```cpp
class histogram {
public:
    void insert(const std::string& s);
```

Private functions

```
class histogram {
public:
    void insert(const std::string& s);
```


New requirement: Insert and remove words separated by a substring.

Private functions

```
class histogram {
public:
    void insert(const std::string& s);
    void insert(std::string_view s, std::string_view sep);
    void remove(std::string_view s);
    void remove(std::string_view s, std::string_view sep);

    size_t operator[](std::string_view s) const;
    [[nodiscard]] bool empty() const { return words_.empty(); }
    [[nodiscard]] size_t size() const { return words_.empty(); }
    auto begin() const;
    auto end() const;
private:
```

Private functions

```
class histogram {
public:
    void insert(const std::string& s);
    void insert(std::string_view s, std::string_view sep);
    void remove(std::string_view s);
    void remove(std::string_view s, std::string_view sep);

    size_t operator[](std::string_view s) const;
    [[nodiscard]] bool empty() const { return words_.empty(); }
    [[nodiscard]] size_t size() const { return words_.empty(); }
    auto begin() const;
    auto end() const;
private:
    void per_word(std::string_view words, std::string_view sep,
                  auto action);
    ...
};
```

Private functions

```
public:
    void insert(std::string_view s, std::string_view sep) {
```

Private functions

```
public:
    void insert(std::string_view s, std::string_view sep) {
```

Call action for every substring

Private functions

```
public:
    void insert(std::string_view s, std::string_view sep) {
        per_word(s, sep, [this](std::string_view word) {
            insert(std::string(word));
        });
    }
private:
    void per_word(std::string_view words, std::string_view sep,
                  auto action)
    {
        while (!words.empty()) {
            auto sep_pos = words.find(sep);
            action(words.substr(0, sep_pos));
            if (sep_pos == std::string_view::npos) break;
            words.remove_prefix(sep_pos + sep.length());
        }
    }
```

Private functions

```
public:
    void insert(std::string_view s, std::string_view sep) {
        per_word(s, sep, [this](std::string_view word) {
            insert(std::string(word));
        });
    }
private:
    void per_word(std::string_view words, std::string_vie        ep,
                  auto action)
    {
        while (!words.empty()) {
            auto sep_pos = words.find(sep);
            action(words.substr(0, sep_pos));
            if (sep_pos == std::string_view::n          eak;
            words.remove_prefix(sep_pos + sep.     ngth());
        }
    }
```

How to test something private?

Private functions

```
public:
    void inser                    td::string_view sep) {
        per_w                     tring_view word) {
            i
        });
    }
private:
    void per_w                    ds, std::string_view sep,

    {
        while (      ds.empty()) {
            auto sep_pos = words.find(sep);
            action(words.substr(0, sep_pos));
            if (sep_pos == std::string_view::npos) break;
            words.remove_prefix(sep_pos + sep.length());
        }
    }
```

Break out the logic into something free standing.

Test it separately.

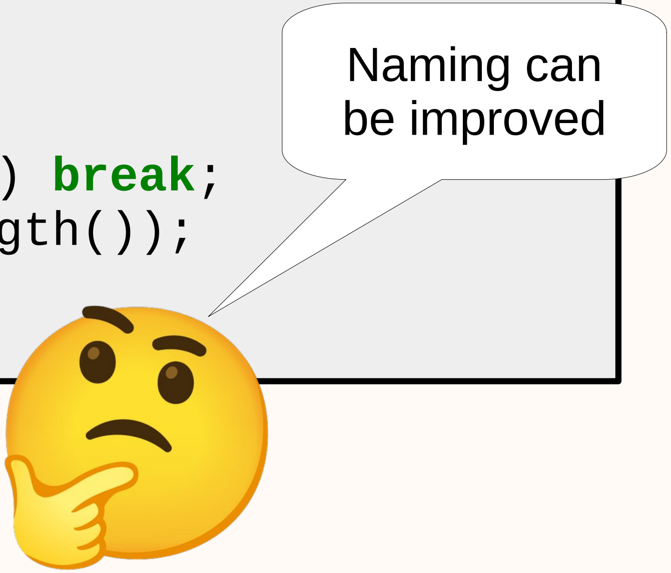
Make your use of the logic a private matter.

Private functions

```
void per_word(std::string_view words, std::string_view sep,
              auto action)
{
    if (sep.empty() || words.empty()) return;
    for (;;) {
        auto sep_pos = words.find(sep);
        action(words.substr(0, sep_pos));
        if (sep_pos == std::string_view::npos) break;
        words.remove_prefix(sep_pos + sep.length());
    }
}
```

Private functions

```cpp
void per_word(std::string_view words, std::string_view sep,
              auto action)
{
    if (sep.empty() || words.empty()) return;
    for (;;) {
        auto sep_pos = words.find(sep);
        action(words.substr(0, sep_pos));
        if (sep_pos == std::string_view::npos) break;
        words.remove_prefix(sep_pos + sep.length());
    }
}
```



Private functions

```cpp
void for_each_word(std::string_view words,
                   std::string_view separator,
                   auto action)
{
    if (separator.empty() || words.empty()) return;
    for (;;) {
        auto sep_pos = words.find(separator);
        action(words.substr(0, sep_pos));
        if (sep_pos == std::string_view::npos) break;
        words.remove_prefix(sep_pos + separator.length());
    }
}
```

Private functions

```
using Catch::Matchers::RangeEquals;

TEST_CASE("action is called on each substring between separators")
{
    std::vector<std::string_view> substrings;
    for_each_word("a,cd,efg", ",", [&](auto w){ substrings.push_back(w); });
    REQUIRE_THAT(substrings, RangeEquals(std::array{"a", "cd", "efg"}));
}
```

Private functions

```
using Catch::Matchers::RangeEquals;

TEST_CASE("action is called on each substring between separators")
{
    std::vector<std::string_view> substrings;
    for_each_word("a,cd,efg", ",", [&](auto w){ substrings.push_back(w); });
    REQUIRE_THAT(substrings, RangeEquals(std::array{"a", "cd", "efg"}));
}
TEST_CASE("action is called on the whole string if separator is not found")
TEST_CASE("if the string ends on separator, an empty string will be the last")
TEST_CASE("if the string begins on separator, an empty string will be first")
TEST_CASE("adjoining separators yields empty strings")
TEST_CASE("action is never called if separator is an empty string")
TEST_CASE("action is never called if the string of words is empty")
```

Private functions

```
using Catch::Matchers::RangeEquals;

TEST_CASE("action is called on each substring between separators")
{
    std::vector<std::string_view> substrings;
    for_each_word("a,cd,efg", ",", [&](auto w){ s
    REQUIRE_THAT(substrings, RangeEquals(std::arr
}
TEST_CASE("action is called on the whole string i
TEST_CASE("if the string ends on separator, an em
TEST_CASE("if the string begins on separator, an
TEST_CASE("adjoining separators yields empty strings
TEST_CASE("action is never called if separator is an      ty string")
TEST_CASE("action is never called if the string of  words is empty")
```

The intended functionality of `for_each_word` is now easy for anyone to see, and its correctness is automatically verified on every run of the test

Private functions

```
#include "for_each_word.h"

class histogram {
public:
    void insert(const std::string& s);
    void insert(std::string_view words, std::string_view separator) {
        for_each_word(words, separator,
                      [this](auto w){insert(std::string(w));});
    }
    void remove(std::string_view s);
    void remove(std::string_view words, std::string_view separator) {
        for_each_word(words, separator, [this](auto w){remove(w);});
    }
    ...
private:
};
```

Private functions

```
#include "for_each_word.h"

class histogram {
public:
    void insert(const std::string& s);
    void insert(std::string_view words                  tor) {
        for_each_word(words, separator
                     [this](auto w){in
    }
    void remove(std::string_view s);
    void remove(std::string_view words,   d::string_view separator) {
        for_each_word(words, separat  r, [this](auto w){remove(w);});
    }
    ...
private:
};
```

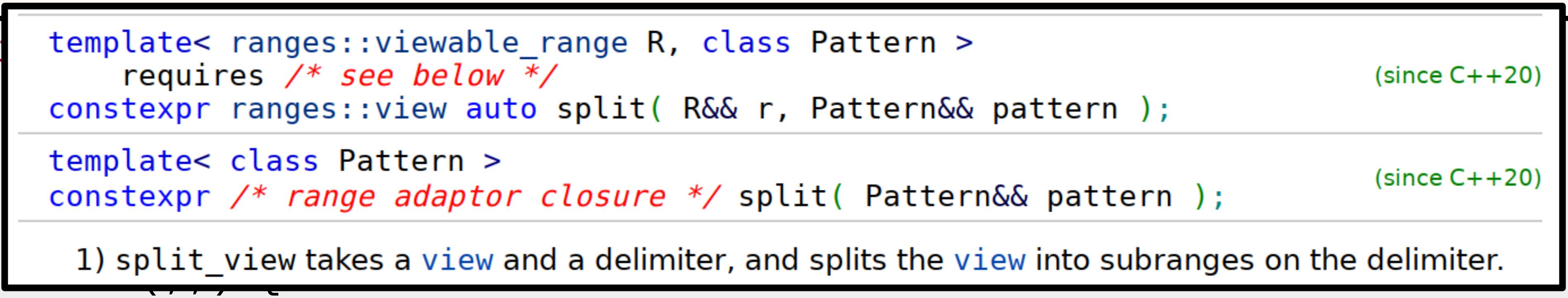
Now you don’t have to test every corner case of `for_each_word`, you test the situations that are interesting for the user of `histogram`.

Refactoring

```cpp
void for_each_word(std::string_view words,
                   std::string_view separator,
                   auto action)
{
    if (separator.empty() || words.empty()) return;
    for (;;) {
        auto sep_pos = words.find(separator);
        action(words.substr(0, sep_pos));
        if (sep_pos == std::string_view::npos) break;
        words.remove_prefix(sep_pos + separator.length());
    }
}
```

Refactoring

```
vo

{

        auto sep_pos = words.find(separator);
        action(words.substr(0, sep_pos));
        if (sep_pos == std::string_view::npos) break;
        words.remove_prefix(sep_pos + separator.length());
    }
}
```

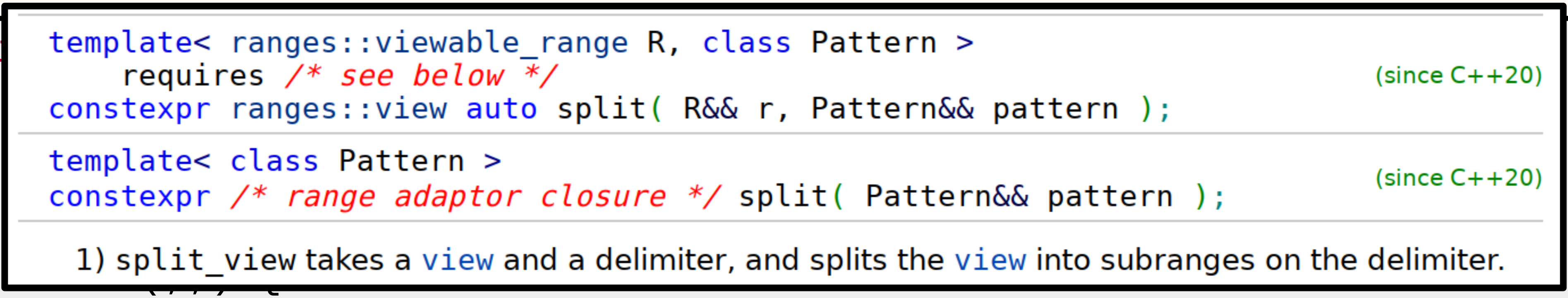


```
template< ranges::viewable_range R, class Pattern >
    requires /* see below */                                        (since C++20)
constexpr ranges::view auto split( R&& r, Pattern&& pattern );

template< class Pattern >                                           (since C++20)
constexpr /* range adaptor closure */ split( Pattern&& pattern );
```

1) split_view takes a view and a delimiter, and splits the view into subranges on the delimiter.

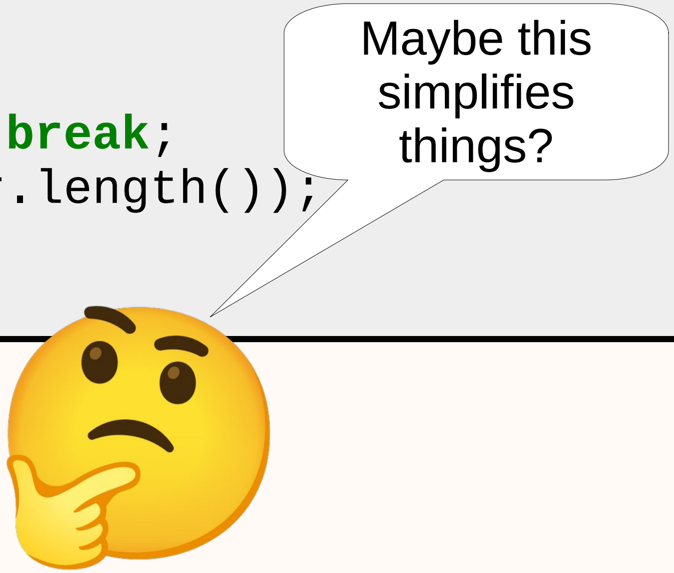
Refactoring

```
vo

{

        auto sep_pos = words.find(separator);
        action(words.substr(0, sep_pos));
        if (sep_pos == std::string_view::npos) break;
        words.remove_prefix(sep_pos + separator.length());
    }
}
```



```
template< ranges::viewable_range R, class Pattern >
    requires /* see below */                                        (since C++20)
constexpr ranges::view auto split( R&& r, Pattern&& pattern );

template< class Pattern >                                           (since C++20)
constexpr /* range adaptor closure */ split( Pattern&& pattern );
```

1) split_view takes a view and a delimiter, and splits the view into subranges on the delimiter.



Refactoring

```
void for_each_word(std::string_view words,
                   std::string_view separator,
                   auto action)
{
    for (auto char_range : std::views::split(words, separator))
    {
        action(std::string_view(char_range));
    }
}
```

Refactoring

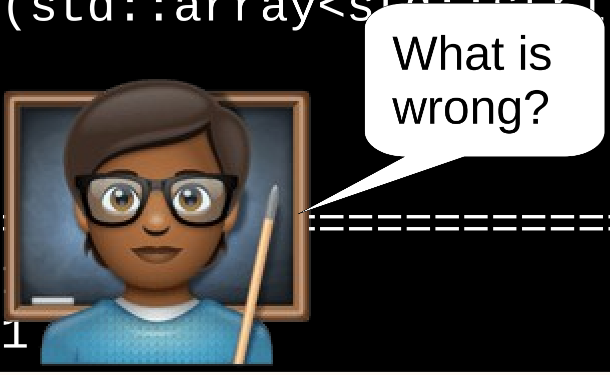
```cpp
void for_each_word(std::string_view words,
                   std::string_view separator,
```

```
-------------------------------------------------------------------
for_each_word does nothing if separator is an empty string
-------------------------------------------------------------------
for_each_word_test.cpp:41
...................................................................
for_each_word_test.cpp:45: FAILED:
  REQUIRE_THAT( substrings,
                RangeEquals(std::array<std::string_view, 0>{}) )
with expansion:
  { "a", "b", "c", "d", "e", "f" }
elements are {  }
===================================================================
test cases: 7 | 6 passed | 1 failed
assertions: 7 | 6 passed | 1 failed
```

Refactoring

```
void for_each_word(std::string_view words,
                   std::string_view separator,
```

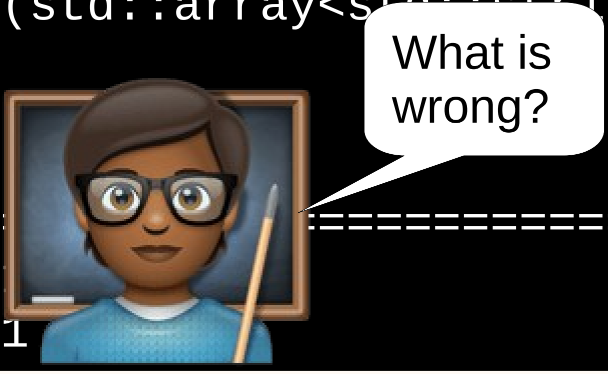
```
-------------------------------------------------------------------------------
for_each_word does nothing if separator is an empty string
-------------------------------------------------------------------------------
for_each_word_test.cpp:41
...............................................................................
for_each_word_test.cpp:45: FAILED:
  REQUIRE_THAT( substrings,
                RangeEquals(std::array<std::string_view, 0>{}) )
with expansion:
  { "a", "b", "c", "d", "e"
elements are {  }
===============================================================================
test cases: 7 | 6 passed | 1
assertions: 7 | 6 passed | 1
```



Refactoring

```
void for_each_word(std::string_view words,
                   std::string_view separator,
```

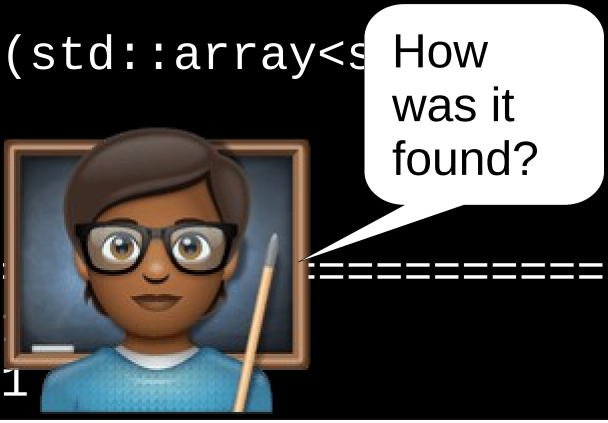
```
-------------------------------------------------------------------
for_each_word does nothing if separator is an empty string
-------------------------------------------------------------------
for_each_word_test.cpp:41
...................................................................
for_each_word_test.cpp:45: FAILED:
  REQUIRE_THAT( substrings,
                RangeEquals(std::array<std::string_view, 0>{}) )
with expansion:
  { "a", "b", "c", "d", "e"
elements are {  }
===================================================================
test cases: 7 | 6 passed |
assertions: 7 | 6 passed | 1
```



Refactoring

```
void for_each_word(std::string_view words,
                   std::string_view separator,
```

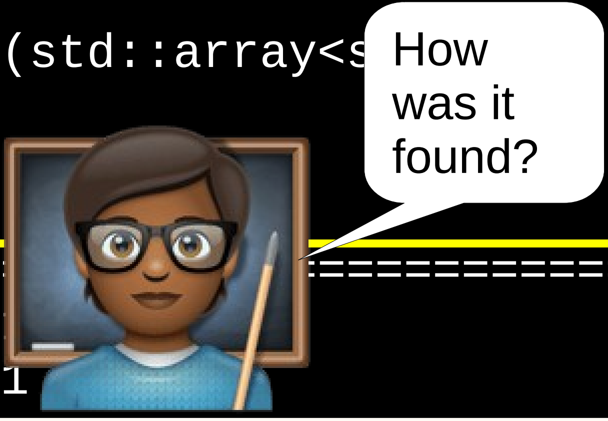
```
-------------------------------------------------------------------
for_each_word does nothing if separator is an empty string
-------------------------------------------------------------------
for_each_word_test.cpp:41
...................................................................
for_each_word_test.cpp:45: FAILED:
  REQUIRE_THAT( substrings,
                RangeEquals(std::array<s          ng_view, 0>{}) )
with expansion:
  { "a", "b", "c", "d", "e"
elements are {  }
===================================================================
test cases: 7 | 6 passed | 1
assertions: 7 | 6 passed | 1
```



Refactoring

```
void for_each_word(std::string_view words,
                   std::string_view separator,
```

```
-------------------------------------------------------------------------------
for_each_word does nothing if separator is an empty string
-------------------------------------------------------------------------------
for_each_word_test.cpp:41
...............................................................................
for_each_word_test.cpp:45: FAILED:
  REQUIRE_THAT( substrings,
               RangeEquals(std::array<s        ng_view, 0>{}) )
with expansion:
  { "a", "b", "c", "d", "e"
elements are {  }
===============================================================================
test cases: 7 | 6 passed | 1
assertions: 7 | 6 passed | 1
```



Refactoring

```
void for_each_word(std::string_view words,
                   std::string_view separator,
                   auto action)
{
    if (separator.empty()) return;
    for (auto char_range : std::views::split(words, separator))
    {
        action(std::string_view(char_range));
    }
}
```

Refactoring

```
void for_each_word(std::string_view words,
                   std::string_view separator,
                   auto action)
{
    if (separator.empty()) return;
    for (auto char_range : std::views::split(words, separator))
    {
        action(std::string_view(char_range));
    }
}
```

```
===============================================================================
All tests passed (7 assertions in 7 test cases)
```

Refactoring

```cpp
void for_each_word(std::string_view words,
                   std::string_view separator,
                   auto action)
{
    if (separator.empty()) return;
    for (auto char_range : std::views::split(words, separator))
    {
        action(std::string_view(char_range));
    }
}
```

```
=====================================================================
All tests passed (7 assertions in 7 test cases)
```

Contracts

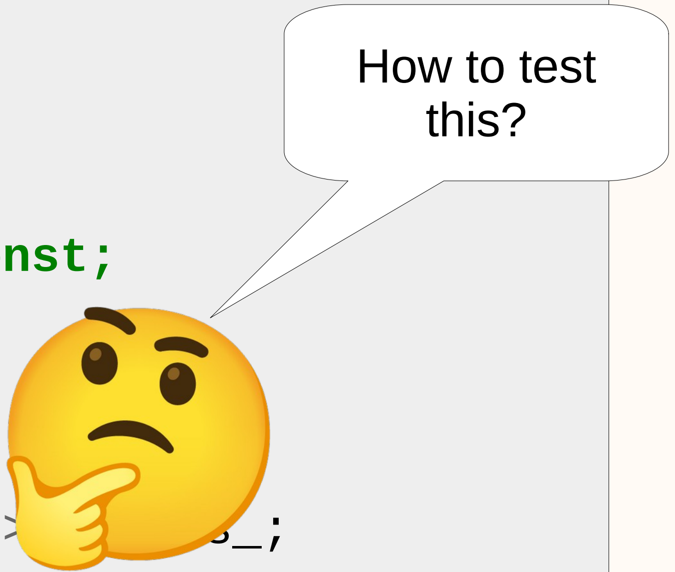
```
class histogram {
public:
    void insert(const std::string& s);

    void remove(std::string_view s) {
        auto iter = words_.find(s);
        assert(iter != words_.end());
        if (--iter->second == 0) {
            words_.erase(iter);
        }
    }
    size_t operator[](std::string_view s) const;

    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```
class histogram {
public:
    void insert(const std::string& s);

    void remove(std::string_view s) {
        auto iter = words_.find(s);
        assert(iter != words_.end());
        if (--iter->second == 0) {
            words_.erase(iter);
        }
    }
    size_t operator[](std::string_view s) const;

    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```
class histogram {
public:
    void insert(const std::string& s);

    void remove(std::string_view s) {
        auto iter = words_.find(s);
        assert(iter != words_.end());
        if (--iter->second == 0) {
            words_.erase(iter);
        }
    }
    size_t operator[](std::string_view s) const;

    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<      s_;
};
```



Contracts

```
class histogram {
```

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

TEST(remove, removing_non_existing_word_is_illegal)
{
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    ASSERT_EXIT(h.remove("banana"),
                          testing::KilledBySignal(SIGABRT), "");
}
```

```
    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```
class histogram {
```

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

TEST(remove, removing_non_existing_word_is_illegal)
{
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    ASSERT_EXIT(h.remove("banana"),
                                 testing::KilledBySignal(SIGABRT), "");
}
```

Another test framework - googletest

```
    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```
class histogram {
```

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

TEST(remove, removing_non_existing_word_is_illegal)
{
    histogram h;
    h.insert("fo
    h.insert("ba
    h.insert("foo");
    h.insert("baz");
    ASSERT_EXIT(h.remove("banana"),
                          testing::KilledBySignal(SIGABRT), "");
}
```

Test suite

```
    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```cpp
class histogram {
```

```cpp
#include <gtest/gtest.h>

TEST(remove, removing_non_existing_word_is_illegal)
{
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.ins
    h.insert("baz");
    ASSERT_EXIT(h.remove("banana"),
                                     testing::KilledBySignal(SIGABRT), "");
}
```

Test case name

```cpp
    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```
class histogram {
```

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

TEST(remove, removing_non_existing_word_is_illegal)
{
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    ASSERT_EXIT(h.remove("banana"),
                          testing::KilledBySignal(SIGABRT), "");
}
```

These must be valid identifiers, hence the ugly formatting

```
    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```
class histogram {
```

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

TEST(remove, removing_non_existing_word_is_illegal)
{
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    ASSERT_EXIT(h.remove("banana"),
                          testing::KilledBySignal(SIGABRT), "");
}
```

```
    auto begin() const;
    auto end() const;
private:
    std::map<std::string                                    s_;
};
```

Google test has “death tests”, which ensures that a call kills the process. It does this by spawning a child process.

Contracts

```
class histogram {
```

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

TEST(remove, removing_non_existing_word_is_illegal)
{
    histogram h;
    h.insert("foo");
    h.insert("bar");
    h.insert("foo");
    h.insert("baz");
    ASSERT_EXIT(h.remove("banana"), testing::KilledBySignal(SIGABRT), "");
}
```

```
    size_t operator[](std::string_view s) const;

    auto begin() const;
    auto end() const;
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```
class histogram {
```

```
#inc

TEST
{

                                                                        ');
}
```

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from remove
[ RUN      ] remove.removing_non_existing_word_is_illegal
htest.cpp:15: Failure
Death test: h.remove("banana")
    Result: died but not with expected exit code:
            Terminated by signal 11 (core dumped)
Actual msg:
[  DEATH   ]
[  FAILED  ] remove.removing_non_existing_word_is_illegal (125 ms)
[----------] 1 test from remove (125 ms total)
[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (125 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] remove.removing_non_existing_word_is_illegal

1 FAILED TEST
```

```
  pr

  };
```

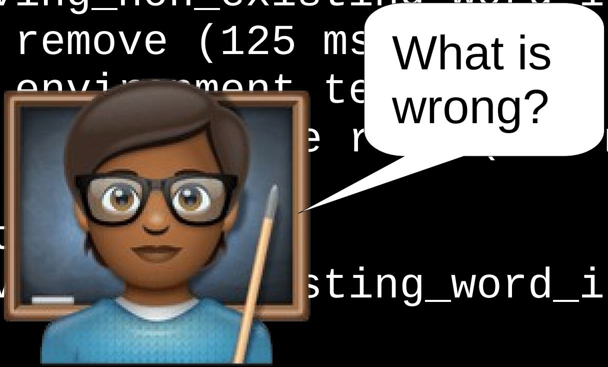
Contracts

```
class histogram {
```

```
#inc

TEST
{

                                                                                          ');
}
```

```
pr

};
```

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from remove
[ RUN      ] remove.removing_non_existing_word_is_illegal
htest.cpp:15: Failure
Death test: h.remove("banana")
    Result: died but not with expected exit code:
            Terminated by signal 11 (core dumped)
Actual msg:
[  DEATH   ]
[  FAILED  ] remove.removing_non_existing_word_is_illegal (125 ms)
[----------] 1 test from remove (125 ms
[----------] Global test environment te
[==========] 1 test from              e r          ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, list
[  FAILED  ] remove.remov          sting_word_is_illegal

1 FAILED TEST
```



Contracts

```
class histogram {
```

```
#inc

TEST
{

    
}
```

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from remove
[ RUN      ] remove.removing_non_existing_word_is_illegal
htest.cpp:15: Failure
Death test: h.remove("banana")
    Result: died but not with expected exit code:
            Terminated by signal 11 (core dumped)
Actual msg:
[  DEATH   ]
[  FAILED  ] remove.removing_non_existing_word_is_illegal (125 ms)
[----------] 1 test from remove (125 ms
[----------] Global test envi      ment te
[==========] 1 test from            e r         ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, list
[  FAILED  ] remove.remov           sting_word_is_illegal

1 FAILED TEST
```

```
');
```

```
pr

};
```

Contracts

```
class histogram {
```

```
#inc

TEST
{



                                                                                 ');
}
```

```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from remove
[ RUN      ] remove.removing_non_existing_word_is_illegal
htest.cpp:15: Failure
Death test: h.remove("banana")
    Result: died but not with expected exit code:
            Terminated by signal 11 (core dumped)
Actual msg:
[  DEATH   ]
[  FAILED  ] remove.removing_non_existi         s_illegal (125 ms)
[----------] 1 test from remove (125 ms
[----------] Global test envi          te
[==========] 1 test from           e r          ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, list
[  FAILED  ] remove.remov           sting_word_is_illegal

1 FAILED TEST
```

```
pr

};
```

Contracts

```
class histogram {
#inc
TEST
{
                                                                                ');
}
  pr
};
```

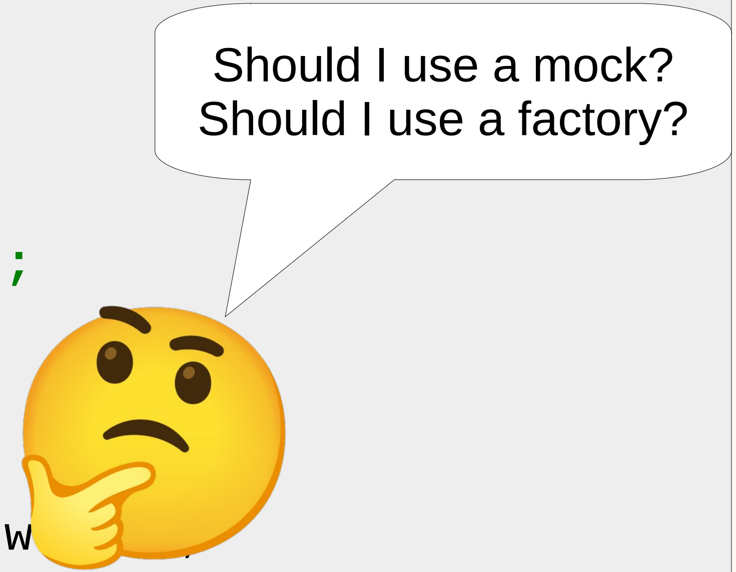
```
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from remove
[ RUN      ] remove.removing_non_existing_word_is_illegal
htest.cpp:15: Failure
Death test: h.remove("banana")
    Result: died but not with expected exit code:
            Terminated by signal 11 (core dumped)
Actual msg:
[  DEATH   ]
[  FAILED  ] remove.removing_non_existing_word_is_illegal (125 ms)
[----------] 1 test from remove (125 ms total)
[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (125 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] remove.removing_non_existing_word_is_illegal

1 FAILED TEST
```

Contracts

```
class histogram {
public:

    void insert(const std::string& s);
    void remove(std::string_view s) {
        auto iter = words_.find(s);
        assert(iter != words_.end());
        if (--iter->second == 0) {
            words_.erase(iter);
        }
    }
    size_t operator[](std::string_view s) const;
    auto begin() const;
    auto end() const;

private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```
class histogram {
public:

    void insert(const std::string& s);
    void remove(std::string_view s) {
        auto iter = words_.find(s);
        assert(iter != words_.end());
        if (--iter->second == 0) {
            words_.erase(iter);
        }
    }
    size_t operator[](std::string_view s) const;
    auto begin() const;
    auto end() const;

private:
    std::map<std::string, size_t, std::less<>> w
};
```



Contracts

```
class histogram {
public:

    void insert(const std::string& s);
    void remove(std::string_view s) {
        auto iter = words_.find(s);
        assert(iter != words_.end());
        if (--iter->second == 0) {
            words_.erase(iter);
        }
    }
    size_t operator[](std::string_view s) const;
    auto begin() const;
    auto end() const;

private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```
class histogram {
public:

    void insert(const std::string& s);
    void remove(std::string_view s) {
        auto iter = words_.find(s);
        assert(iter != words_.end());
        if (--iter->second == 0) {
            words_.erase(iter);
        }
    }
    size_t operator[](std::string_view s) const;
    auto begin() const;
    auto end() const;

private:
    std::map<std::string, size_t, std::less<>> w
};
```

Alternatively…?

Contracts

```
class histogram {
public:
    using precondition = void(*)();
    void insert(const std::string& s);
    void remove(std::string_view s, precondition p = default_handler) {
        auto iter = words_.find(s);
        assert(iter != words_.end());
        if (--iter->second == 0) {
            words_.erase(iter);
        }
    }
    size_t operator[](std::string_view s) const;
    auto begin() const;
    auto end() const;
    static void default_handler() { abort(); }
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```
class histogram {
public:
    using precondition = void(*)();
    void insert(const std::string& s);
    void remove(std::string_view s, precondition p = default_handler) {
        auto iter = words_.find(s);
        assert(iter != words_.end());
        if (--iter->second == 0) {
            words_.erase(iter);
        }
    }
    size_t operator[](std::string_view s) const;
    auto begin() const;
    auto end() const;
    static void default_handler() { abort(); }
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```
class histogram {
public:
    using precondition = void(*)();
    void insert(const std::string& s);
    void remove(std::string_view s, precondition p = default_handler) {
        auto iter = words_.find(s);
        assert(iter != words_.end());
        if (--iter->second == 0) {
            words_.erase(iter);
        }
    }
    size_t operator[](std::string_view s) const;
    auto begin() const;
    auto end() const;
    static void default_handler() { abort(); }
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```
class histogram {
public:
    using precondition = void(*)();
    void insert(const std::string& s);
    void remove(std::string_view s, precondition p = default_handler) {
        auto iter = words_.find(s);
        assert(iter != words_.end());
        if (--iter->second == 0) {
            words_.erase(iter);
        }
    }
    size_t operator[](std::string_view s) const;
    auto begin() const;
    auto end() const;
    static void default_handler() { abort(); }
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```
class histogram {
public:
    using precondition = void(*)();
    void insert(const std::string& s);
    void remove(std::string_view s, precondition p = default_handler) {
        auto iter = words_.find(s);
        assert(iter != words_.end());
        if (--iter->second == 0) {
            words_.erase(iter);
        }
    }
    size_t operator[](std::string_view s) const;
    auto begin() const;
    auto end() const;
    static void default_handler() { abort(); }
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```
class histogram {
public:
    using precondition = void(*)();
    void insert(const std::string& s);
    void remove(std::string_view s, precondition p = default_handler) {
        auto iter = words_.find(s);
        assert(iter != words_.end());
        if (--iter->second == 0) {
            words_.erase(iter);
        }
    }
    size_t operator[](std::string_view s) const;
    auto begin() const;
    auto end() const;
    static void default_handler() { abort(); }
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```
class histogram {
public:
    using precondition = void(*)();
    void insert(const std::string& s);
    void remove(std::string_view s, precondition p = default_handler) {
        auto iter = words_.find(s);
        if (iter == words_.end()) p();
        if (--iter->second == 0) {
            words_.erase(iter);
        }
    }
    size_t operator[](std::string_view s) const;
    auto begin() const;
    auto end() const;
    static void default_handler() { abort(); }
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```
class histogram {
p
```

```
TEST_CASE("Removing a non-existing word is illegal")
{
    struct test_exception {};
    histogram h;
    for (auto word : {"foo","bar","foo","baz"})
    {
        h.insert(word);
    }
    REQUIRE_THROWS_AS(h.remove("banana",
                               +[](){ throw test_exception{};}),
                      test_exception);
}
```

```
                                                                           {
    auto end() const;
    static void default_handler() { abort(); }
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```
class histogram {
p
```

```
TEST_CASE("Removing a non-existing word is illegal")
{
    struct test_exception {};
    histogram h;
    for (auto word : {"foo","bar","foo","baz"})
    {
        h.insert(word);
    }
    REQUIRE_THROWS_AS(h.remove("banana",
                               +[](){ throw test_exception{};}),
                      test_exception);
}
```

Back to catch2

```
{
    auto end() const;
    static void default_handler() { abort(); }
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```
class histogram {
p
                                                                              {
    auto end() const;
    static void default_handler() { abort(); }
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

```
TEST_CASE("Removing a non-existing word is illegal")
{
    struct test_exception {};
    histogram h;
    for (auto word : {"foo","bar","foo","baz"})
    {
        h.insert(word);
    }
    REQUIRE_THROWS_AS(h.re
                                +[](){ throw test_exception{};}),
                      test_exception);
}
```

Back to catch2

An exception type that can only exist in the scope of this test case

Contracts

```
class histogram {
p
                                                                                                   {
    auto end() const;
    static void default_handler() { abort(); }
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

```
TEST_CASE("Removing a non-existing word is illegal")
{
    struct test_exception {};
    histogram h;
    for (auto wor                  oo","baz"})
    {
        h.insert(
    }
    REQUIRE_THROWS_AS(h.remove("banana",
                               +[](){ throw test_exception{};}),
                      test_exception);
}
```

Back to catch2

A custom precondition handler that throws our exception type

Contracts

```
class histogram {
p
                                                                                {
    auto end() const;
    static void default_handler() { abort(); }
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

```
TEST_CASE("Removing a non-existing word is illegal")
{
    struct test_exception
    histogram h;
    for (auto word : {"foo                                 )
    {
        h.insert(word);
    }
    REQUIRE_THROWS_AS(h.remove("banana",
                               +[](){ throw test_exception{};}),
                      test_exception);
}
```

Ensure that removing a non-existing word throws our exception

Back to catch2

Contracts

```
class histogram {
p
  TEST
  {
                                                                                          {

  }
```

```
------------------------------------------------------------------
Removing a non-existing word is illegal
------------------------------------------------------------------
htest.cpp:120
..................................................................
htest.cpp.cpp:128: FAILED:
  REQUIRE_THROWS_AS( h.remove("banana",
     +[](){ throw test_exception{};}), test_exception )
because no exception was thrown where one was expected:
==================================================================
test cases: 5 | 4 passed | 1 failed
assertions: 8 | 7 passed | 1 failed
```

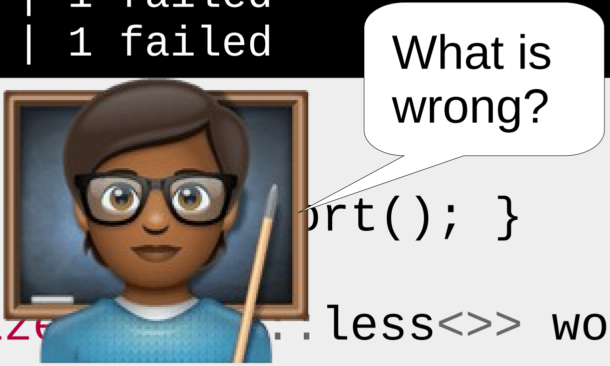
```
    auto begin() const;
    auto end() const;
    static void default_handler() { abort(); }
private:
    std::map<std::string, size_t, std::less<>> words_;
};
```

Contracts

```
class histogram {
p  TEST
   {
```

```
----------------------------------------------------------------------
Removing a non-existing word is illegal
----------------------------------------------------------------------
htest.cpp:120
......................................................................
htest.cpp.cpp:128: FAILED:
  REQUIRE_THROWS_AS( h.remove("banana",
     +[](){ throw test_exception{};}), test_exception )
because no exception was thrown where one was expected:
======================================================================
test cases: 5 | 4 passed | 1 failed
assertions: 8 | 7 passed | 1 failed
```

```
   }
    auto begin() const;
    auto end() const;
    static void default_hand                    rt(); }
private:
    std::map<std::string, siz            ::less<>> words_;
};
```



Contracts

```
class histogram {
p TEST
  {
```

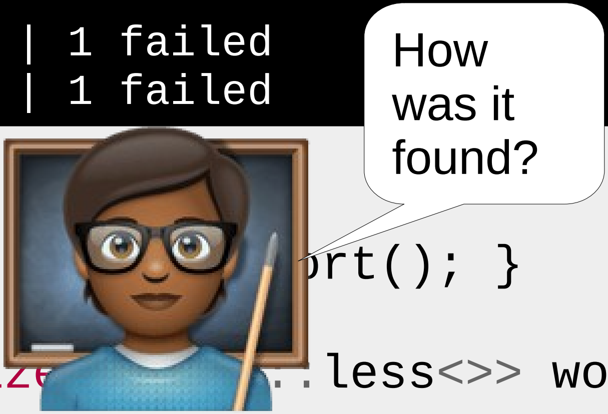
```
--------------------------------------------------------------
Removing a non-existing word is illegal
--------------------------------------------------------------
htest.cpp:120
..............................................................
htest.cpp.cpp:128: FAILED:
  REQUIRE_THROWS_AS( h.remove("banana",
     +[](){ throw test_exception{};}), test_exception )
because no exception was thrown where one was expected:
==============================================================
test cases: 5 | 4 passed | 1 failed
assertions: 8 | 7 passed | 1 failed
```

```
  }
    auto begin() const;
    auto end() const;
    static void default_hand                 rt(); }
private:
    std::map<std::string, size             ::less<>> words_;
};
```

Contracts

```
class histogram {
p
  TEST
  {
                                                                                    {

  }
    auto begin() const;
    auto end() const;
    static void default_hand                  rt(); }
private:
    std::map<std::string, siz                ::less<>> words_;
};
```

```
-------------------------------------------------------------------
Removing a non-existing word is illegal
-------------------------------------------------------------------
htest.cpp:120
...................................................................
htest.cpp.cpp:128: FAILED:
  REQUIRE_THROWS_AS( h.remove("banana",
     +[](){ throw test_exception{};}), test_exception )
because no exception was thrown where one was expected:
===================================================================
test cases: 5 | 4 passed | 1 failed
assertions: 8 | 7 passed | 1 failed
```



Contracts

```
class histogram {
p
```

```
TEST
{



}
```

```
------------------------------------------------------------------
Removing a non-existing word is illegal
------------------------------------------------------------------
htest.cpp:120
..................................................................
htest.cpp.cpp:128: FAILED:
  REQUIRE_THROWS_AS( h.remove("banana",
     +[](){ throw test_exception{};}), test_exception )
because no exception was thrown where one was expected:
==================================================================
test cases: 5 | 4 passed | 1 failed
assertions: 8 | 7 passed | 1 failed
```

```
    auto begin() const;
    auto end() const;
    static void default_hand                 rt(); }
private:
    std::map<std::string, size          ::less<>> words_;
};
```

Contracts

```
class histogram {
public:
    using precondition = void(*)();
    void insert(const std::string& s);
    void remove(std::string_view s, precondition p = default_handler) {
        auto iter = words_.find(s);
        if (iter == words_.end()) p();
        if (--iter->second == 0) {
            words_.erase(iter);
        }
    }
    void remove(std::string_view words, std::string_view separator);
    size_t operator[](std::string_view s) const;
    auto begin() const;
    auto end() const;
    static void default_handler() { abort(); }
    ...
};
```

Contracts

```
class histogram {
public:
    using precondition = void(*)();
    void insert(const std::string& s);
    void remove(std::string_vie                    = default_handler) {
        auto iter = words_.find
        if (iter == words_.end(
        if (--iter->second == 0)
            words_.erase(iter);
        }
    }
    void remove(std::string_view words, std::string_view separator);
    size_t operator[](std::string_view s) const;
    auto begin() const;
    auto end() const;
    static void default_handler() { abort(); }
    ...
};
```

How about this?

Contracts

```
class histogram {
public:
    using precondition = void(*)();
    void insert(const std::string& s);
    void remove(std::stri                               p = default_handler) {
        auto iter = wo
        if (iter == wo
        if (--iter->se
            words_.era
        }
    }
    void remove(std::string_view words,       string_view separator);
    size_t operator[](std::string_view s) con   ;
    auto begin() const;
    auto end() const;
    static void default_handler() { abort(); }
    ...
};
```

Difficulty to express a test case is often a hint that something is problematic with the design.

In this case it is the API that is problematic.

Redesign

```
class histogram {
public:
    void insert(const std::string& s);
    size_t remove(std::string_view s) {
        auto iter = words_.find(s);
        if (iter == words_.end()) return 0;
        if (--iter->second == 0) {
            words_.erase(iter);
        }
        return 1;
    }
    size_t remove(std::string_view words, std::string_view separator);
    size_t operator[](std::string_view s) const;
    auto begin() const;
    auto end() const;
    static void default_handler() { abort(); }
    ...
};
```

Redesign

```
class histogram {
public:
    void insert(const std::string& s);
    size_t remove(std::string_view s) {
        auto iter = words_.find(s);
        if (iter == words_.end()) return 0;
        if (--iter->second == 0) {
            words_.erase(iter);
        }
        return 1;
    }
    size_t remove(std::string_view words, std::string_view separator);
    size_t operator[](std::string_view s) const;
    auto begin() const;
    auto end() const;
    static void default_handler() { abort(
    ...
};
```

Maybe the precondition was a mistake, and it's better to return the number of words removed?

The caller can decide how they want to deal with it.

Redesign

```
class histogram {
public:
    void insert(const std::string& s);
    size_t remove(std::string_view s) {
        auto iter = words_.find(s);
        if (iter == words_.end()) return 0;
        if (--iter->second == 0) {
            words_.erase(iter);
        }
        return 1;
    }
    size_t remove(std::string_view words, std::stri g_view separator);
    size_t operator[](std::string_view s) const;
    auto begin() const;
    auto end() const;
    static void default_handler() { abort(
    ...
};
```

Maybe the precondition was a mistake, and it's better to return the number of words removed?

The caller can decide how they want to deal with it.

Redesign

```
class histogram {
public:
    void insert(const std::string& s);
    size_t remove(std::string_view s) {
        auto iter = words_.find(s);
        if (iter == words_.end()) return 0;
        if (--iter->second == 0) {
            words_.erase(iter);
        }
        return 1;
    }
    size_t remove(std::string_view words, std::string_view separator);
    size_t operator[](std::string_view s) const;
    auto begin() const;
    auto end() const;
    static void default_handler() { abort(); }
    ...
};
```

Redesign

```
class histogram {
p
};
```

```
SCENARIO("Removing words") {
    GIVEN("a histogram with some words") {
        histogram h;
        for (auto word : {"foo", "bar", "baz","banana", "baz"}) {
            h.insert(word);
        }
        WHEN("removing a word not in the histogram") {
            THEN("0 is returned") {
                REQUIRE(h.remove("boo") == 0);
            }
        }
        AND_WHEN("removing a word in the histogram") {
            THEN("1 is returned") {
                REQUIRE(h.remove("baz"));
            }
        }
        AND_WHEN("removing multiple semicolon separated words")
        ...
```

Redesign

```
class histogram {
p
  SCEN
        }
      }
      AND_WHEN("removing multiple semicolon separated words")
      ...
};
```

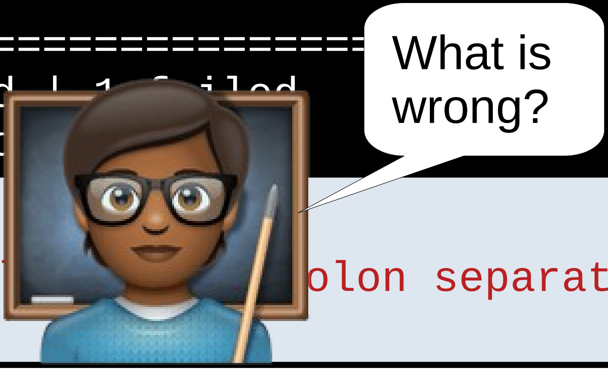
```
-----------------------------------------------------------------------
Scenario: Removing words
      Given: a histogram with some words
   And when: removing multiple semicolon separated words
       Then: The number of existing words that were removed is returned
----------------------------------------------------------------------
htest.cpp:143
.......................................................................
htest.cpp:145: FAILED:
  REQUIRE( num_removed == 2 )
with expansion:  3 == 2
=======================================================================
test cases:  5 | 4 passed | 1 failed
assertions: 10 | 9 passed | 1 failed
```

Redesign

```
class histogram {
p  SCEN
```

```
--------------------------------------------------------------
Scenario: Removing words
      Given: a histogram with some words
   And when: removing multiple semicolon separated words
       Then: The number of existing words that were removed is returned
--------------------------------------------------------------
htest.cpp:143
..............................................................
htest.cpp:145: FAILED:
  REQUIRE( num_removed == 2 )
with expansion:  3 == 2
==============================================================
test cases:  5 | 4 passed | 1 failed
assertions: 10 | 9 passed
```

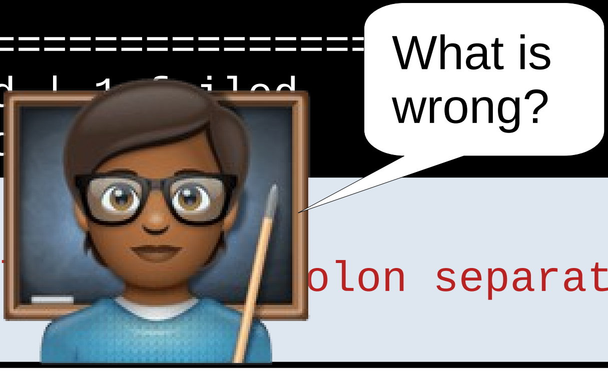
```
            }
        }
        AND_WHEN("removing mu            olon separated words")
        ...
};
```



Redesign

```
class histogram {
p  SCEN





           }
       }
       AND_WHEN("removing mu          olon separated words")
       ...
};
```

```
Scenario: Removing words
      Given: a histogram with some words
   And when: removing multiple semicolon separated words
       Then: The number of existing words that were removed is returned
----------------------------------------------------------------------
htest.cpp:143
......................................................................
htest.cpp:145: FAILED:
  REQUIRE( num_removed == 2 )
with expansion:  3 == 2
======================================================================
test cases:  5 | 4 passed | 1 failed
assertions: 10 | 9 passed
```



Redesign

```
class histogram {
p  SCEN
```

```
-------------------------------------------------------------------------------
Scenario: Removing words
      Given: a histogram with some words
   And when: removing multiple semicolon separated words
       Then: The number of existing words that were removed is returned
-------------------------------------------------------------------------------
htest.cpp:143
...............................................................................
htest.cpp:145: FAILED:
  REQUIRE( num_removed == 2 )
with expansion:  3 == 2
===============================================================================
test cases:  5 | 4 passed | 1 failed
assertions: 10 | 9 passed
```

```
            }
        }
        AND_WHEN("removing mu          olon separated words")
        ...
};
```

Redesign

```
class histogram {
p  SCEN

          }
      }
      AND_WHEN("removing mu          olon separated words")
      ...
};
```

```
-------------------------------------------------------------------------------
Scenario: Removing words
      Given: a histogram with some words
   And when: removing multiple semicolon separated words
       Then: The number of existing words that were removed is returned
-------------------------------------------------------------------------------
htest.cpp:143
...............................................................................
htest.cpp:145: FAILED:
  REQUIRE( num_removed == 2 )
with expansion:  3 == 2
===============================================================================
test cases:  5 | 4 passed | 1 failed
assertions: 10 | 9 passed
```

Contracts

```
class histogram {
public:
    using precondition = void(*)();
    void insert(const std::string& s);
    void remove(std::string_view s, precondition p = default_handler) {
        auto iter = words_.find(s);
        if (iter == words_.end()) p();
        if (--iter->second == 0) {
            words_.erase(iter);
        }
    }
    void remove(std::string_view words, std::string_view separator);
    size_t operator[](std::string_view s) const;
    auto begin() const;
    auto end() const;
    static void default_handler() { abort(); }
    ...
};
```

Contracts

```
class histogram {
public:
    using precondition = void(*)();
    void insert(const std::string& s);
    void remove(std::string_view s, precondition p = default_handler) {
        auto iter = words_.find(s);
        if (iter == words_.end()) p();
        if (--iter->second == 0) {
            words_.erase(iter);
        }
    }
    void remove(std::string_view words, std:      ing_view separator);
    size_t operator[](std::string_view s)  nst;
    auto begin() const;
    auto end() const;
    static void default_handler()        t(); }
    ...
};
```

Or maybe this whole API extension was a mistake and we should let the caller do the loop and make the decision?

Contracts

```
class histogram {
public:
```

Or maybe this whole API extension was a mistake and we should let the caller do the loop and make the decision?

Contracts

```
class histogram {
public:
```

```
void user_class::user_func() {
    for_each_word(words_, ";",
                  [this](auto w){
                      if (histogram_[w] > 0) {
                          histogram_.remove(std::string(w));
                      }
                  });
}
```

Wrapping up

Wrapping up

- A good test tells you what’s wrong by telling you what right would be.

Wrapping up

- A good test tells you what’s wrong by telling you what right would be.
- A good test works as documentation that is automatically verified.

Wrapping up

- A good test tells you what’s wrong by telling you what right would be.
- A good test works as documentation that is automatically verified.
- A good test suite is a set of very simple tests.

Wrapping up

- A good test tells you what’s wrong by telling you what right would be.
- A good test works as documentation that is automatically verified.
- A good test suite is a set of very simple tests.
- Listen to your code. Difficulty to express a test is your code’s way of saying that something is problematic with it.

Wrapping up

- A good test tells you what’s wrong by telling you what right would be.
- A good test works as documentation that is automatically verified.
- A good test suite is a set of very simple tests.
- Listen to your code. Difficulty to express a test is your code’s way of saying that something is problematic with it.
- Beware of dogmas.

Wrapping up

- A good test tells you what’s wrong by telling you what right would be.
- A good test works as documentation that is automatically verified.
- A good test suite is a set of very simple tests.
- Listen to your code. Difficulty to express a test is your code’s way of saying that something is problematic with it.
- Beware of dogmas.
  - Though many serve as good guidelines.

Wrapping up

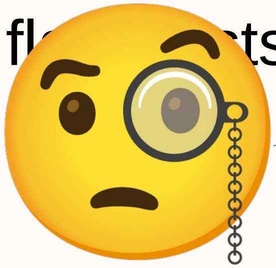
- A good test tells you what’s wrong by telling you what right would be.
- A good test works as documentation that is automatically verified.
- A good test suite is a set of very simple tests.
- Listen to your code. Difficulty to express a test is your code’s way of saying that something is problematic with it.
- Beware of dogmas.
  - Though many serve as good guidelines.
- A bad test is superior to no test.

Wrapping up

- A good test tells you what’s wrong by telling you what right would be.
- A good test works as documentation that is automatically verified.
- A good test suite is a set of very simple tests.
- Listen to your code. Difficulty to express a test is your code’s way of saying that something is problematic with it.
- Beware of dogmas.
  - Though many serve as good guidelines.
- A bad test is superior to no test.
  - Except flaky tests. Never accept flaky tests.

Wrapping up

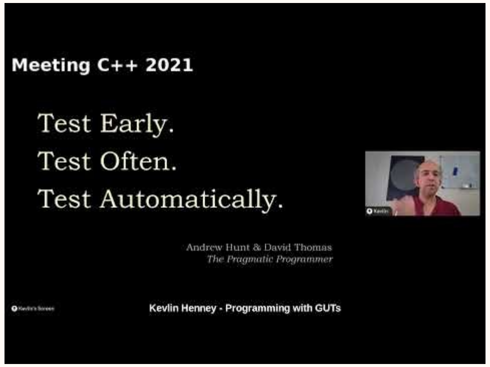
- A good test tells you what’s wrong by telling you what right would be.
- A good test works as documentation that is automatically verified.
- A good test suite is a set of very simple tests.
- Listen to your code. Difficulty to express a test is your code’s way of saying that something is problematic with it.
- Beware of dogmas.
  - Though many serve as good guidelines.
- A bad test is superior to no test.
  - Except flaky tests. Never accept fl ts.



Software tends to evolve for much longer than you think.

Tests help!

Thanks

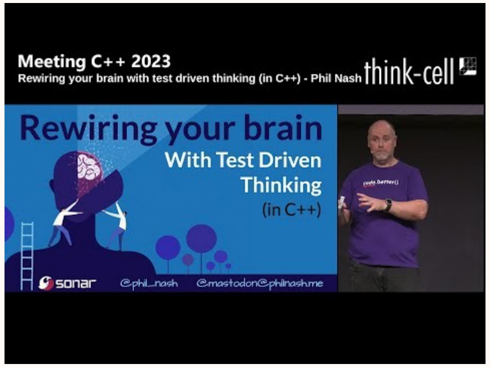


Meeting C++ 2021

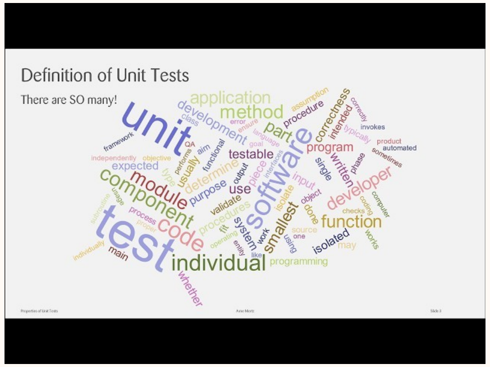
Test Early.
Test Often.
Test Automatically.

Andrew Hunt & David Thomas
The Pragmatic Programmer

Kevlin Henney - Programming with GUTs



Meeting C++ 2023
Rewiring your brain with test driven thinking (in C++) - Phil Nash



Definition of Unit Tests

There are SO many!

Kevlin Henney
Programming with GUTs

https://youtu.be/cfh6ZrA19r4

Phil Nash
Rewiring Your Brain with
Test Driven Thinking in C++

https://youtu.be/Hx-1Wtvhvgw

Arne Mertz
Properties of Unit Tests in C++

https://youtu.be/Ko4r-rixZVk

Will your program still be correct next year?

Björn Fahller

bjorn@fahller.se

@rollbear.bsky.social

@rollbear@fosstodon.org

@rollbear

#include <C++>