cppcon
the c++ conference

Mix Assertion, Logging, Unit Testing and Fuzzing with ZeroErr

Build Safer Modern C++ Application

Speaker: Xiaofan Sun
Date: Sep 19, 2024

NVIDIA

Self-Introduction

- Got my Ph.D. from UC, Riverside last year
- Automatic testing of multithreading programs
  - Symbolic execution improvements
  - Fault detection in concurrent data structures
- Now working in NVIDIA HWInf Team
- Job: Infrastructure toolchains for Registers

Xiaofan Sun

NVIDIA

Motivation

The Story about ZeroErr framework

Two Years Ago...

```
A list of types needed to print:

Custom Classes:                 struct myStruct;
Custom containers:              myOrderedMap<std::string, myStruct>
Smart Pointer:                  std::unique_ptr<myStruct>
Class from third-party library: llvm::Value*
```

Logging the Data

```
// LOG(INFO) << Data;
// ASSERT(a > b, “A > B is not true”, a, b);

std::ostream& operator<<(std::ostream& out, const myStruct& data);
std::ostream& operator<<(std::ostream& out, my_ordered_map<std::string, myStruct> data);
std::ostream& operator<<(std::ostream& out, std::unique_ptr<myStruct> ptr);
std::ostream& operator<<(std::ostream& out, llvm::Value* data);
......
```

Logging the Data

- Namespace pollution
- Hard to implement with template
- No extensibility
- No customization for different scenario for the same type

A better way is using a formatting-like interface, and a stateful functor:

```
LOG("Input is: {}", input);     →     zeroerr::format(__VA_ARGS__)
```

```
template <typename... T>
std::string operator()(const char* fmt, T&&... args)
```

Printing in both Logging & Assertion

Logging macro:

```
        LOG("Input is: {}", input);
```

Smart assertion in user code / unit testing also need pretty printing, e.g.:

```
        ASSERT(a != 0, "a should not be 0. Input is: {}", input);
```

Do I catch the bug?

Yes, and it’s inside a unit test case.

Then, why the unit test case passed?

Another Issue

How can I check the cache worked?

```
Expr* parseExpr(std::string input)
{
    static std::map<std::string, Expr*> cache;
    if (cache.count(input) == 0) {
        Expr* expr = parse_the_input(input)
        cache[input] = expr;
        return expr;
    } else {
        return cache[input]->Clone();
    }
}

TEST_CASE("parsing test") {
    Expr* e1 = parseExpr("1 + 2");
    Expr* e2 = parseExpr("1 + 2 ");
    // Some checks for e1 and e2
}
```

There is a bug in Clone

Did you see the space?

Another Issue

Access log message can give additional safety for unit testing.

```
Expr* parseExpr(std::string input)
{
    static std::map<std::string, Expr*> cache;
    if (cache.count(input) == 0) {
        Expr* expr = parse_the_input(input)
        cache[input] = expr;
        return expr;
        LOG("Cache hit for input: ", input);
        return cache[input]->Clone();
    }
}

TEST_CASE("parsing test") {
    Expr* e1 = parseExpr("1 + 2");
    Expr* e2 = parseExpr("1 + 2 ");
    // TODO: check the cache worked
}   CHECK(LOG_GET() == "Cache hit for input: 1 + 2");
}
```

Error Code VS Check Log

```
int foo() {
    if (error1_occurred) { return 1; }
    if (error2_occurred) { return 2; }
    // some implementation
    return 0;
}
```

```
bool foo() {
    if (error1_occurred) {
        LOG("Error 1 occurred due to some reason, "
            "the current value of X, Y, Z is: ", X, Y, Z);
        return false;
    }
    if (error2_occurred) {
        LOG("Error 2 occurred due to some reason, "
            "the current value of X, Y, Z is: ", X, Y, Z);
        return false;
    }
    // some implementation
    return true;
}
```

Error Code VS Check Log

There are some benefits for checking the log data

- No need to change the API
- No need to maintain the Error Code
- Can check detailed information for a log message
- Can capture additional context information if needed
- Make sure specific path is taken

Structure-Aware Fuzzing

Generation-based fuzzers usually target a single input type - string. All input is reading from a file which generated. For example, libfuzzer:

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  DoSomethingInterestingWithMyAPI(Data, Size);
  return 0;
}
```

For complex data structure, the generated input should first verify it fit in the data structure, then running the test.

Benefits of Integration

- Fuzzing test case can use all those features
- Fuzzing do not need additional assertion implementation
- Writing fuzzing test case as well as unit test case so that they can share code base

Motivation of ZeroErr

- Providing a way to make logged data can be accessed in unit testing
- No need to write print function for a compositional type (e.g. std::map<std::string, int>)
- Allow user to write assertion for both in source code and unit testing code
- Failure assertion can be logged
- All features provided could be used in fuzzing

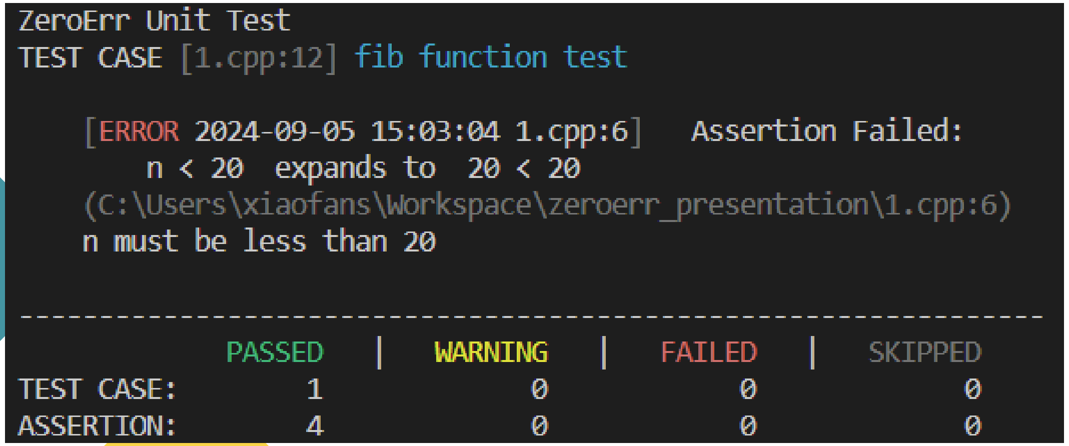
Related Works

ZeroErr is highly influenced by 3 widely-known C++ libraries:

- doctest/doctest: The fastest feature-rich C++11/14/17/20/23 single-header testing framework (github.com)
- sharkdp/dbg-macro: A dbg(...) macro for C++ (github.com)
- google/fuzztest (github.com)

Assertion Example

- How to use the framework



```
ZeroErr Unit Test
TEST CASE [1.cpp:12] fib function test

    [ERROR 2024-09-05 15:03:04 1.cpp:6]   Assertion Failed:
        n < 20  expands to  20 < 20
    (C:\Users\xiaofans\Workspace\zeroerr_presentation\1.cpp:6)
    n must be less than 20


-----------------------------------------------------------------
             PASSED    |   WARNING    |   FAILED    |   SKIPPED
TEST CASE:        1            0             0             0
ASSERTION:        4            0             0             0
```

```cpp
#define ZEROERR_IMPLEMENTATION
#include "zeroerr.hpp"

int fib(int n) {
    REQUIRE(n >= 0, "n must be non-negative");
    REQUIRE(n < 20, "n must be less than 20");
    if (n <= 1) {
        return 1;
    }
    return fib(n - 1) + fib(n - 2);
}

TEST_CASE("fib function test") {
    CHECK(fib(0) == 1);
    CHECK(fib(1) == 1);
    CHECK(fib(2) == 2);
    CHECK_THROWS(fib(20));
}
```

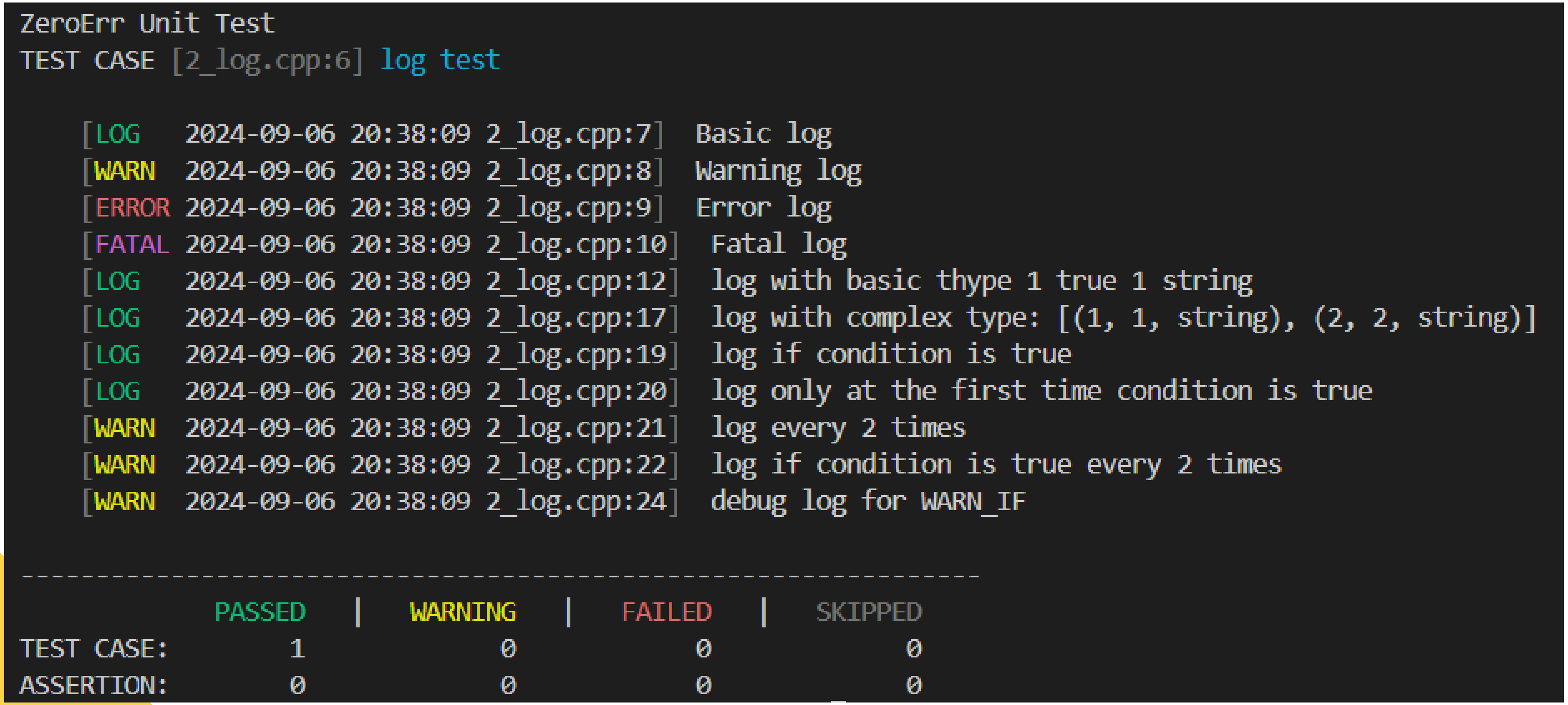
Logging Example

```
TEST_CASE("log test") {
    LOG("Basic log");
    WARN("Warning log");
    ERR("Error log");
    FATAL("Fatal log");
    LOG("log with basic thype {} {} {} {}", 1, true, 1.0, "string");

    std::vector<std::tuple<int, float, std::string>> data = {
        {1, 1.0, "string"}, {2, 2.0, "string"}
    };
    LOG("log with complex type: {data}", data);

    LOG_IF(1==1, "log if condition is true");
    LOG_FIRST(1==1, "log only at the first time condition is true");
    WARN_EVERY_(2, "log every 2 times");
    WARN_IF_EVERY_(2, 1==1, "log if condition is true every 2 times");
    DLOG(WARN_IF, 1==1, "debug log for WARN_IF");
}
```

Logging Example



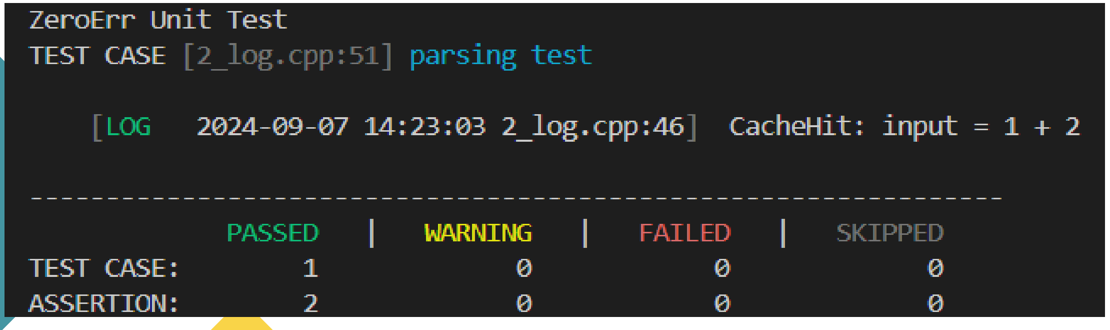
```
ZeroErr Unit Test
TEST CASE [2_log.cpp:6] log test

    [LOG   2024-09-06 20:38:09 2_log.cpp:7]  Basic log
    [WARN  2024-09-06 20:38:09 2_log.cpp:8]  Warning log
    [ERROR 2024-09-06 20:38:09 2_log.cpp:9]  Error log
    [FATAL 2024-09-06 20:38:09 2_log.cpp:10]  Fatal log
    [LOG   2024-09-06 20:38:09 2_log.cpp:12]  log with basic thype 1 true 1 string
    [LOG   2024-09-06 20:38:09 2_log.cpp:17]  log with complex type: [(1, 1, string), (2, 2, string)]
    [LOG   2024-09-06 20:38:09 2_log.cpp:19]  log if condition is true
    [LOG   2024-09-06 20:38:09 2_log.cpp:20]  log only at the first time condition is true
    [WARN  2024-09-06 20:38:09 2_log.cpp:21]  log every 2 times
    [WARN  2024-09-06 20:38:09 2_log.cpp:22]  log if condition is true every 2 times
    [WARN  2024-09-06 20:38:09 2_log.cpp:24]  debug log for WARN_IF

---------------------------------------------------------------
             PASSED    |    WARNING    |    FAILED    |    SKIPPED
TEST CASE:        1              0             0              0
ASSERTION:        0              0             0              0
```

Logging Example

LOG_GET(func, prefix, field, T)
LogStream::getLog<T>(func, prefix, field)



```
ZeroErr Unit Test
TEST CASE [2_log.cpp:51] parsing test

   [LOG   2024-09-07 14:23:03 2_log.cpp:46]  CacheHit: input = 1 + 2

-----------------------------------------------------------------
             PASSED    |   WARNING   |   FAILED   |   SKIPPED
TEST CASE:        1           0             0             0
ASSERTION:        2           0             0             0
```

```cpp
Expr* parseExpr(std::string input)
{
    static std::map<std::string, Expr*> cache;
    if (cache.count(input) == 0) {
        Expr* expr = parse_the_input(input);
        cache[input] = expr;
        return expr;
    } else {
        LOG("CacheHit: input = {input}", input);
        return cache[input]->Clone();
    }
}

TEST_CASE("parsing test") {
    zeroerr::suspendLog();
    std::string log;
    Expr* e1 = parseExpr("1 + 2");
    log = LOG_GET(parseExpr, "CacheHit", input, std::string);
    CHECK(log == std::string{});
    Expr* e2 = parseExpr("1 + 2");
    log = zeroerr::LogStream::getDefault()
        .getLog<std::string>("parseExpr", "CacheHit", "input");
    CHECK(log == "1 + 2");
    zeroerr::resumeLog();
}
```

Fuzzing Example

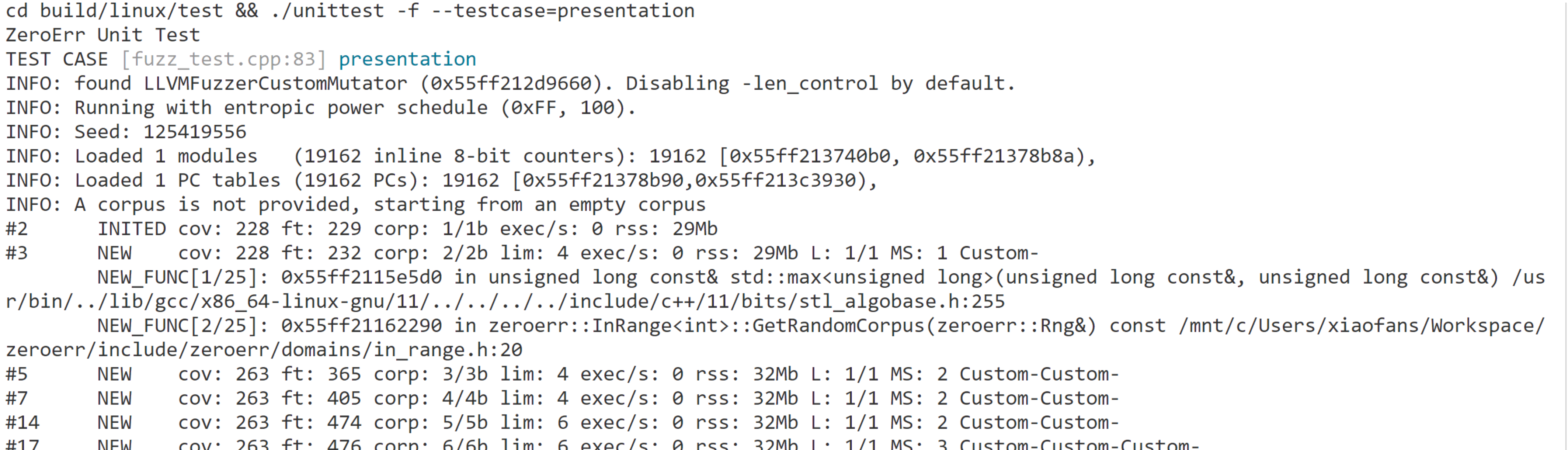
```
unsigned find_the_biggest(const std::vector<unsigned>& vec) {
    if (vec.empty()) {
        WARN("Empty vector, vec.size() = {size}", vec.size());
        return 0;
    }
    // implementation
}
```

```
FUZZ_TEST_CASE("fuzz_test") {
    FUZZ_FUNC([=](const std::vector<unsigned>& vec) {
        zeroerr::suspendLog();
        unsigned ans = find_the_biggest(vec);
        // verify the result
        for (unsigned i = 0; i < vec.size(); ++i) CHECK(ans >= vec[i]);
        if (vec.size() == 0) {
            CHECK(ans == 0);
            // verify WARN message to make sure the path is correct
            CHECK(LOG_GET(find_the_biggest,
            "Empty vector, vec.size() = {size}", size, size_t) == 0);
        }
        zeroerr::resumeLog();
    })
        .WithDomains(ContainerOf<std::vector<unsigned>>(InRange<unsigned>(0, 100)))
        .WithSeeds({{{0, 1, 2, 3, 4, 5}}, {{1, 8, 4, 2, 3}}})
        .Run(100);
}
```

Fuzzing Example

Using Clang and libFuzzer:

clang++ -std=c++11 -fsanitize=**fuzzer-no-link** -L=`clang++ -print-runtime-dir` **-lclang_rt.fuzzer_no_main-x86_64** -o test_fuzz test_fuzz.cpp

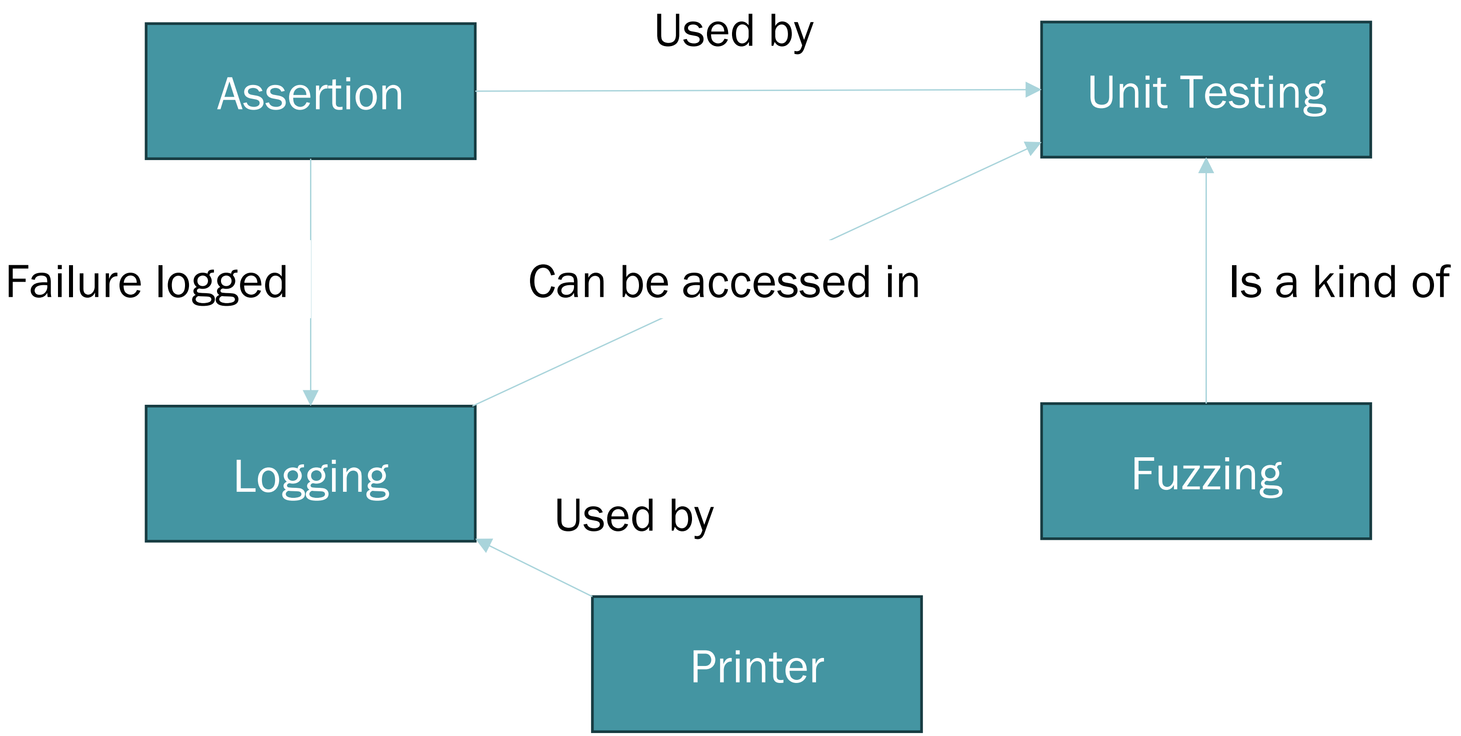


```
cd build/linux/test && ./unittest -f --testcase=presentation
ZeroErr Unit Test
TEST CASE [fuzz_test.cpp:83] presentation
INFO: found LLVMFuzzerCustomMutator (0x55ff212d9660). Disabling -len_control by default.
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 125419556
INFO: Loaded 1 modules   (19162 inline 8-bit counters): 19162 [0x55ff213740b0, 0x55ff21378b8a),
INFO: Loaded 1 PC tables (19162 PCs): 19162 [0x55ff21378b90,0x55ff213c3930),
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 228 ft: 229 corp: 1/1b exec/s: 0 rss: 29Mb
#3      NEW    cov: 228 ft: 232 corp: 2/2b lim: 4 exec/s: 0 rss: 29Mb L: 1/1 MS: 1 Custom-
        NEW_FUNC[1/25]: 0x55ff2115e5d0 in unsigned long const& std::max<unsigned long>(unsigned long const&, unsigned long const&) /us
r/bin/../lib/gcc/x86_64-linux-gnu/11/../../../../include/c++/11/bits/stl_algobase.h:255
        NEW_FUNC[2/25]: 0x55ff21162290 in zeroerr::InRange<int>::GetRandomCorpus(zeroerr::Rng&) const /mnt/c/Users/xiaofans/Workspace/
zeroerr/include/zeroerr/domains/in_range.h:20
#5      NEW    cov: 263 ft: 365 corp: 3/3b lim: 4 exec/s: 0 rss: 32Mb L: 1/1 MS: 2 Custom-Custom-
#7      NEW    cov: 263 ft: 405 corp: 4/4b lim: 4 exec/s: 0 rss: 32Mb L: 1/1 MS: 2 Custom-Custom-
#14     NEW    cov: 263 ft: 474 corp: 5/5b lim: 6 exec/s: 0 rss: 32Mb L: 1/1 MS: 2 Custom-Custom-
#17     NEW    cov: 263 ft: 476 corp: 6/6b lim: 6 exec/s: 0 rss: 32Mb L: 1/1 MS: 3 Custom-Custom-Custom-
```

What makes ZeroErr Different

- Provide a cohesive solution for mixing assertion, logging, unit testing and fuzzing.
- Logged data is structural and accessible
- A structure-aware fuzzing API for quickly create fuzzing test cases as easy as writing unit tests.

Relationship of Components



Assertion

Unit Testing

Logging

Fuzzing

Printer

Used by

Failure logged

Can be accessed in

Is a kind of

Used by

Pretty Printer

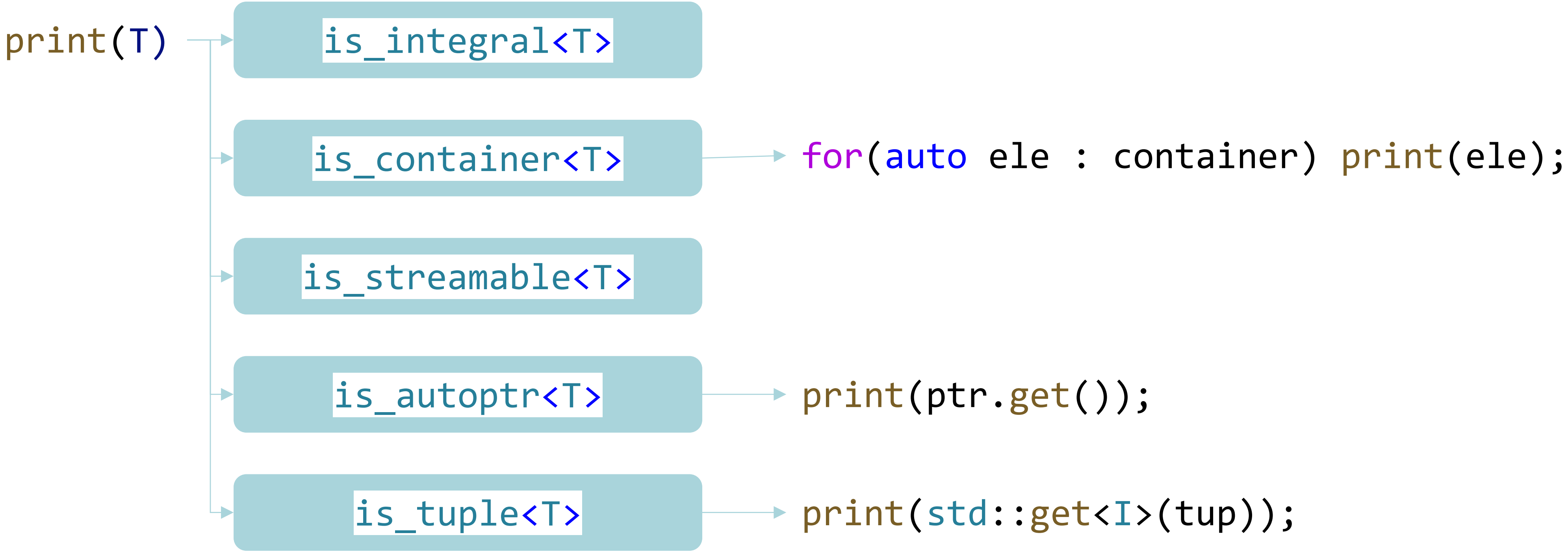
Design & Implementation

Using a template

Assuming type K and V is streamable.

```cpp
template <typename K, typename V>
std::ostream& operator<<(std::ostream& os, const std::map<K, V>& map) {
    os << "{";
    for (auto it = map.begin(); it != map.end(); ++it) {
        os << it->first << ": " << it->second;
        if (std::next(it) != map.end()) {
            os << ", ";
        }
    }
    os << "}";
    return os;
}
```

Decomposing a Type



print(T)

is_integral<T>

is_container<T>

```
for(auto ele : container) print(ele);
```

is_streamable<T>

is_autoptr<T>

```
print(ptr.get());
```

is_tuple<T>

```
print(std::get<I>(tup));
```

How about matching Integral Types

```
template <typename T>
std::enable_if_t<std::is_integral_v<T>, std::ostream&>
operator<<(std::ostream& os, T num) {
    os << num << " (i" << sizeof(T) * 8 << ")";
    return os;
}
```

How about Containers

```
std::vector<int> vec = {1, 2, 3, 4, 5};
```

If we want to print the content of any container.

We will find all containers which have ‘begin()’ and ‘end()’ functions for iterating.

Write Custom Type Traits

```cpp
template <typename... Ts>
using void_t = void;

template <typename T, typename = void>
struct iterable : std::false_type {};

template <typename T>
struct iterable<T,
    void_t<decltype(std::declval<T>().begin()),
           decltype(std::declval<T>().end())>
> : std::true_type {};
```

Write Custom Type Traits (2)

```
template <typename T, typename = void>
struct contain_to_string : std::false_type {};

template <typename T>
struct contain_to_string<T,
    void_t<decltype(std::declval<T>().to_string())>
> : std::true_type {};
```

Conflicts in rules

```
template <typename T>
typename std::enable_if<iterable<T>::value, std::ostream&>::type
operator<<(std::ostream& os, const T& ctn);

template <typename T>
typename std::enable_if<contain_to_string<T>::value, std::ostream&>::type
operator<<(std::ostream& os, const T& obj);
```

What if a class is a container but also provided a custom ‘to_string’ method?
Which rule should be matched?

1 iterable

2 contain_to_string

3 compilation error

Conflicts in rules (2)

```
 // you need to avoid char since it is conflicting with the default implementation
template <typename T>
typename std::enable_if<!std::is_same<T, char>::value, std::ostream&>::type
operator<<(std::ostream& os, const T* ptr) {
    os << typeid(T).name() << "* (" << (void*)ptr << ")";
    return os;
}
```

Problem Definition

- Have a list of (customizable) type traits to decide a template can be enabled
- Apply priority for the rules

Using Overloaded Methods

```
template <unsigned N>
struct rank : rank<N - 1> {};
template <>
struct rank<0> {static_assert(false, "No matching specialization found"); };

template<typename T> std::enable_if_t<std::is_integral<T>::value>
void Foo(T v, rank<1>); // lowest priority
template<typename T> std::enable_if_t<sizeof(T) == 4>
void Foo(T v, rank<2>); // higher priority

template <typename T>
void Foo(T v) { Foo(v, rank<2>{}); }
```

Using Partial Specialization

```
template <typename T, unsigned N = 2, typename = void>
struct foo : foo<T, N-1> {};

template <typename T>
struct foo <T, 0> { static_assert(false, "No matching specialization found"); };

template <typename T>
struct foo <T, 1, enable_if_t<sizeof(T) == 4>>  {
    static void print() { cout << "size 4" << endl; }
};

template <typename T>
struct foo <T, 2, enable_if_t<is_integral<T>::value>>  {
    static void print() { cout << "integral" << endl; }
};
```

Priority 1
Lower

Priority 2
Higher

Assertion

In user code & unit testing

One Assertion – Two Behaviors

In User Source Code:

- Log if failed
- Throw exception

In Unit Test Cases:

- Print if failed
- Count if failed
- Throw exception

Assertion Implementation

Define a special Global variable:

```
namespace {
constexpr bool _ZEROERR_TEST_CONTEXT = false;
}  // namespace
```

Add a parameter in Unit Testing:

```
static void function(zeroerr::TestContext*
                       _ZEROERR_TEST_CONTEXT)
```

Inside the assertion macro implementation:

```
#define ZEROERR_ASSERT_EXP(cond, level, expect_throw, is_false, ...)                             \
        ……                                                                                       \
        zeroerr::detail::context_helper<                                                         \
            decltype(_ZEROERR_TEST_CONTEXT),                                                     \
            std::is_same<decltype(_ZEROERR_TEST_CONTEXT),                                        \
                         const bool>::value>::setContext(assertion_data, _ZEROERR_TEST_CONTEXT); \
```

Assertion in Unit Testing

C++11 Style constexpr if:

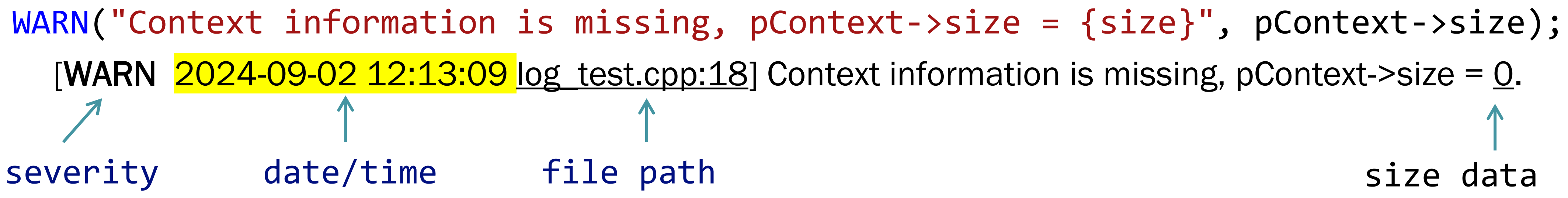
```
template <typename T>
struct context_helper<T, true> {
    static void setContext(AssertionData& data, T) {
       // implementation 1
    }
};

template <typename T>
struct context_helper<T, false> {
    static void setContext(AssertionData& data, T ctx) {
       // implementation 2
    }
};
```

Logging API

Design & Implementation

The Idea of Structure Log



```
WARN("Context information is missing, pContext->size = {size}", pContext->size);
[WARN  2024-09-02 12:13:09 log_test.cpp:18] Context information is missing, pContext->size = 0.
```

Structure Log can be easily accessed since it only stores the variant parts of the log message

Faster and memory friendly since only one memory copy is applied

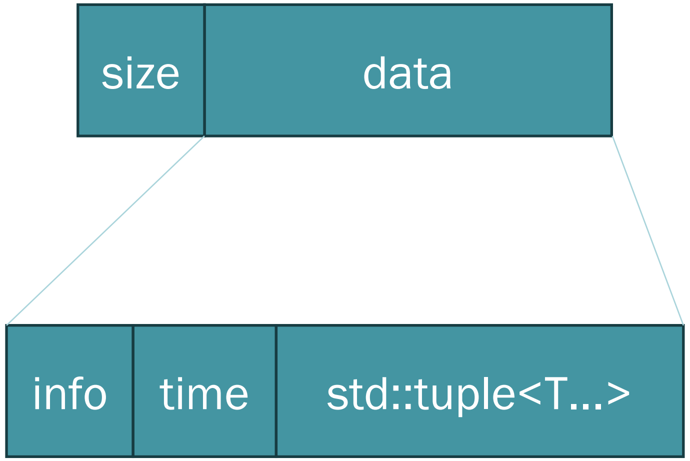
When Object is not Copyable

You may want to stringify the object when you log it:

```
zeroerr::Printer stringify;
LOG("Hello {obj}", stringify(obj));
```

- Manually control when the object is stringified.
- Store the string instead of the value of the object
- You can use other format libraries, e.g. C++20 std::format or {fmt} library

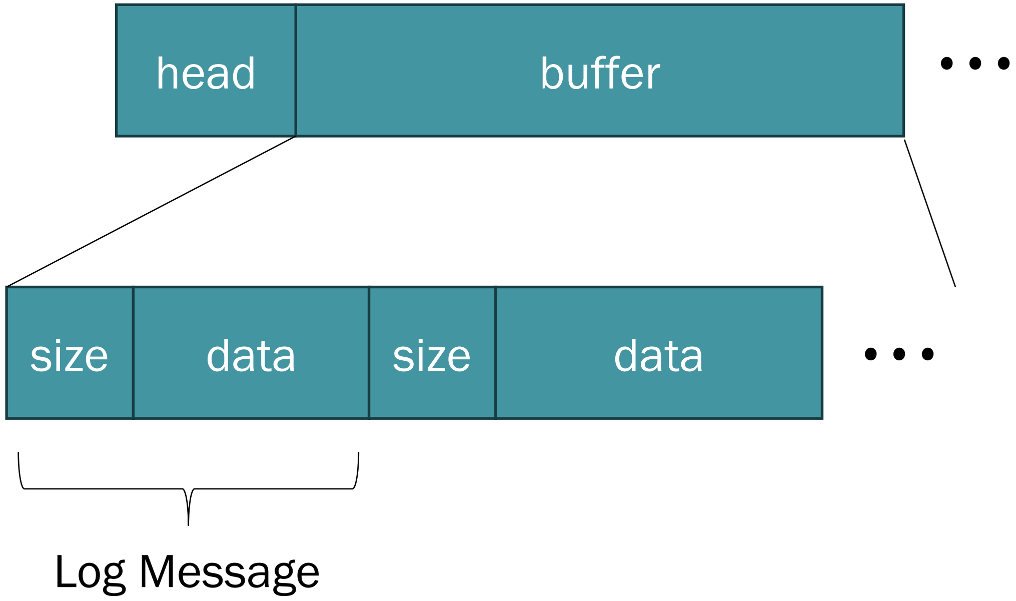
Log Message Data Structure



A Log Message consist of:

- Pointer to log info (file path, message, severity, etc)
- Time
- A tuple of arguments of the message

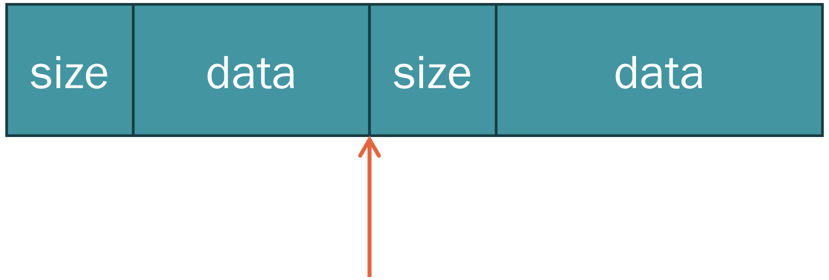
Log Buffer



Buffer block:

1. Head – size, additional information
2. Buffer – a data buffer with multiple log data

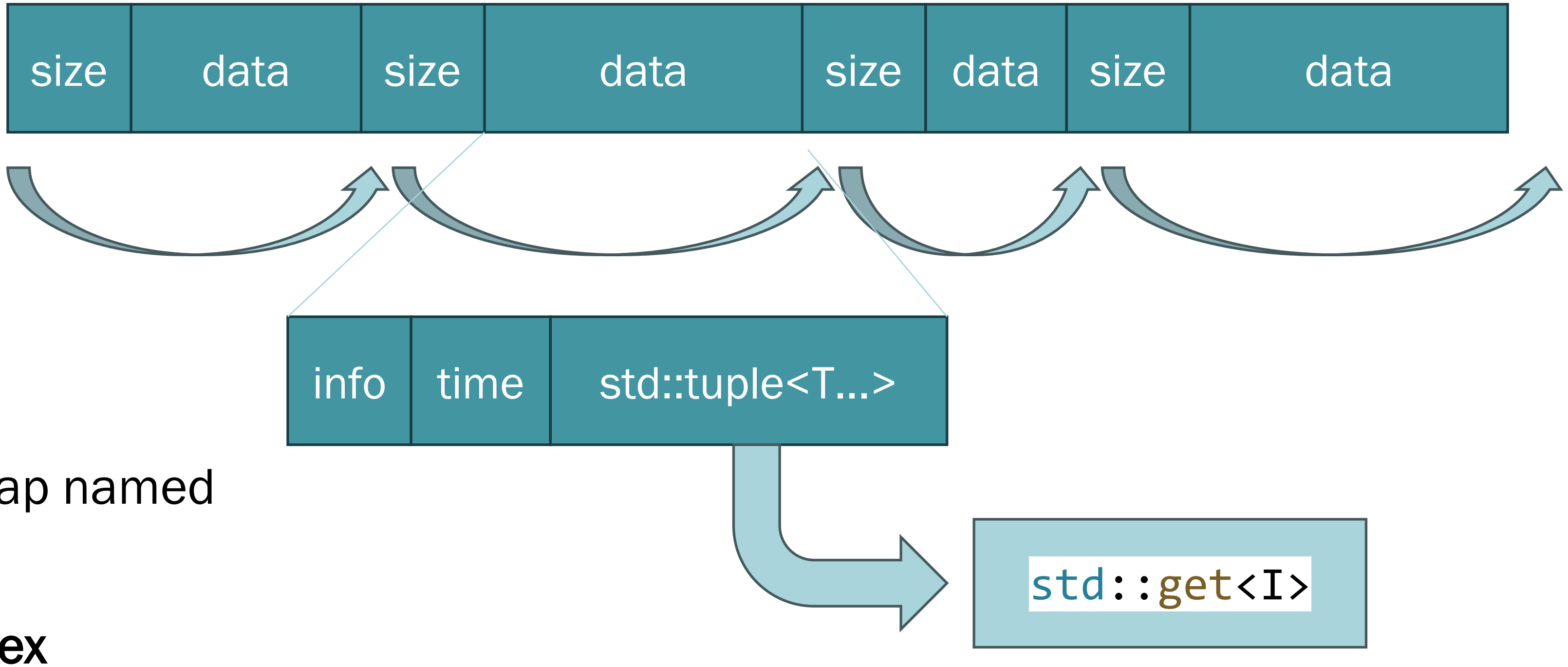
Concurrent queue



There are 3 steps in adding a log message:

- Move the buffer pointer with **atomic add**
- If new position overflow, then handle it with a **lock**
- Then, construct the log message at the original position

Log Iterator



- Jump to data tuple field
- Lookup meta data to map named field into index
- **Get the value at the index**

Visit Tuples using Dynamic Index

```cpp
template <size_t I> struct visit_impl {
    template <typename T, typename F>
    static void visit(T& tup, size_t idx, F&& function) {
        if (idx == I - 1) function(std::get<I - 1>(tup));
        else  visit_impl<I - 1>::visit(tup, idx, std::forward<F>(fun));
    }
};

template <> struct visit_impl<0> {
    template <typename T, typename F> static void visit(T&, size_t, F&&) {}
};

template <typename F, typename... Ts>
void visit_at(const std::tuple<Ts...>& tup, size_t idx, F&& fun) {
    visit_impl<sizeof...(Ts)>::visit(tup, idx, std::forward<F>(fun));
}
```

Fuzzing API

Keynotes & Improvements

Domain & Corpus

Domain is a set of all possible inputs for a data structure

Corpus is the internal representation of a domain

Those two concepts are coming from google/fuzztest and autotest.

Priority of Arbitrary Rules

Arbitrary has similar issues for maintaining a list of rules

```
template <typename T, unsigned N = 2, typename = void>
class Arbitrary : public Arbitrary<T, N-1> {};

template <typename T>
struct Arbitrary <T, 0> {
    static_assert(detail::always_false<T>::value, "No Arbitrary
specialization for this type");
};
```

Priority of Arbitrary Rules

You could write a list of rules with different priorities with this pattern:

```
class Arbitrary<T, 2, is_unsigned_int<T>> : public DomainConvertable<T>
class Arbitrary<T, 2, is_string<T>> : public Domain<T, std::vector<typename T::value_type>>
class Arbitrary<T, 1, is_modifiable<T>>
    : public SequenceContainerOf<T, Arbitrary<typename T::value_type>>
class Arbitrary<std::tuple<T...>, 1>
    : public AggregateOf<std::tuple<typename std::remove_const<T>::type...>>
```

Roadmap

- Decorators – mark/enhance test cases, better integration with log/assertion
- Better printing for large/complex types
- Binary log data output (to reduce log data size)
- Logging/Iterating performance optimization (e.g. using skip-list in log data structure)
- More internal domain types for fuzzing
- More Fuzzing API to control storing/loading corpus
- Allow custom random number generator
- Extensions for other scenarios (e.g. MPI)

Q&A session

Xiaofan Sun

xiaofans@nvidia.com

Online Demo

https://replit.com/@sunxfancy/ZeroErr-Demo

https://github.com/sunxfancy/zeroerr