

CPP-Summit 2019

# 全球C++软件技术大会

C++ Development Technology Summit

**Boolan**

高端IT互联网教育平台



关注“博览Boolan”服务号  
发现更多 会议·课程·活动



# Apex.AI

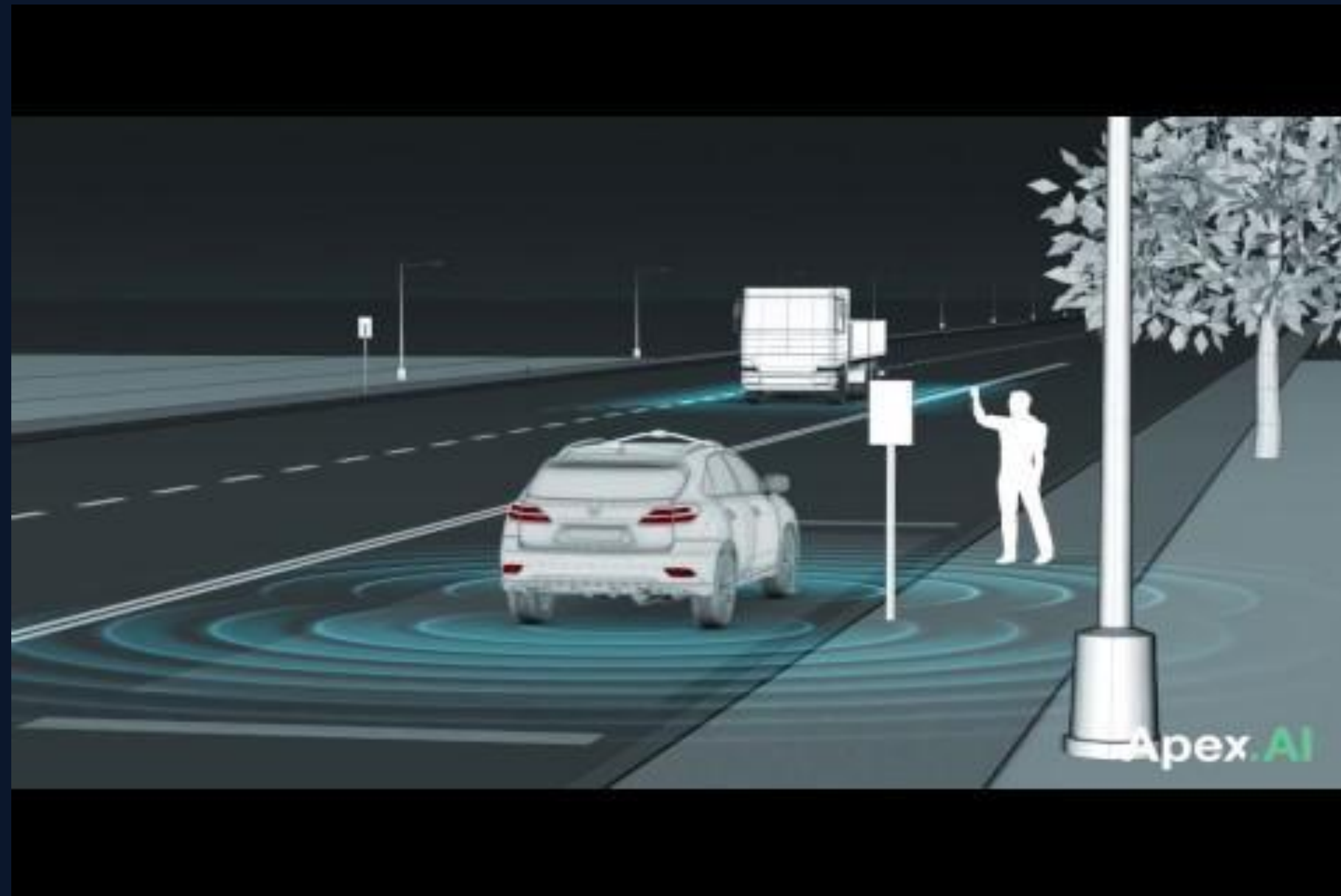
Safe Software for  
Autonomous Mobility With  
Modern C++

Presenter: Andreas Pasternak

# Imagine!

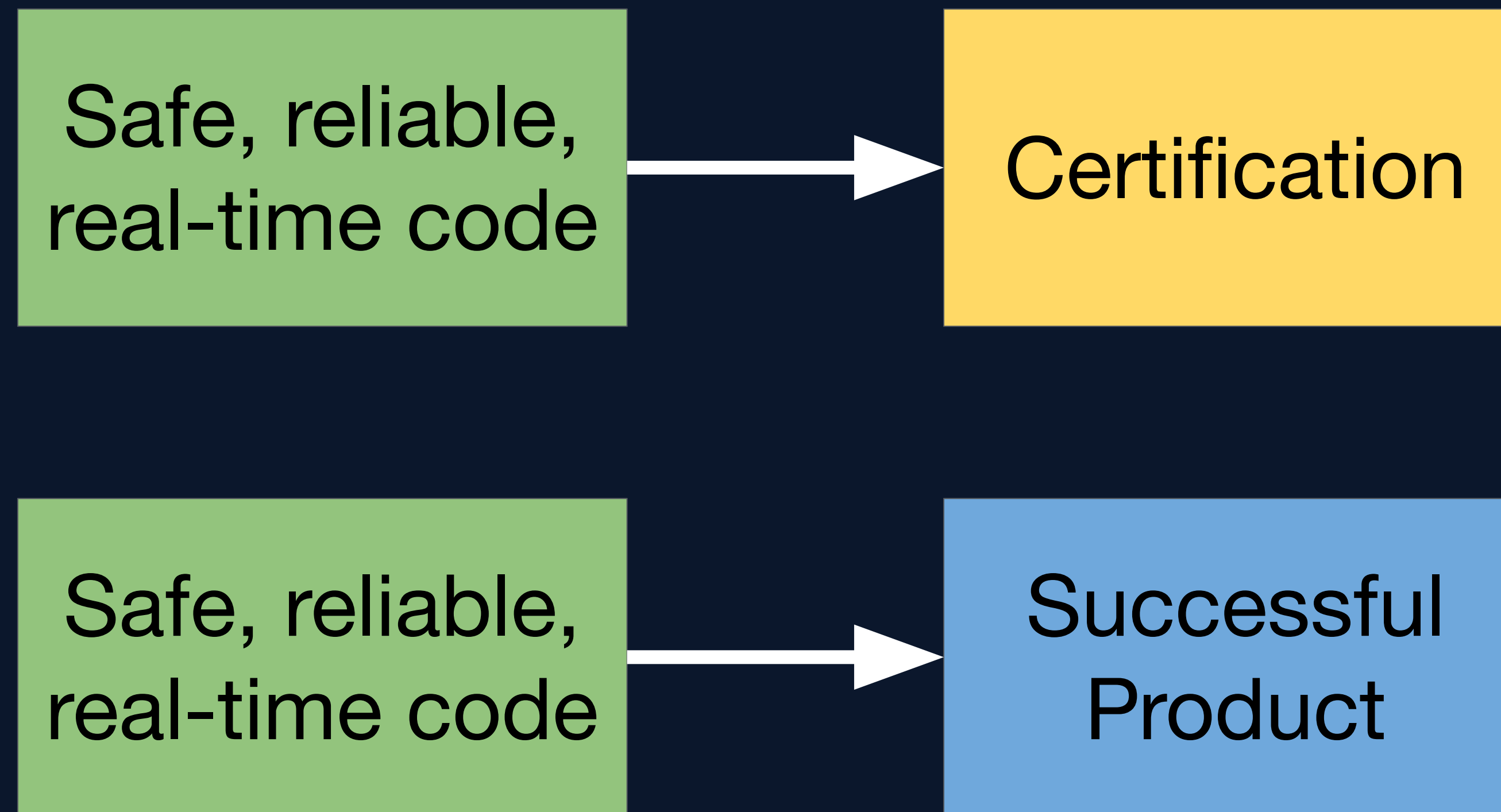
Imagine you are a C++ engineer / lead (maybe domain expert in robotics etc.) and your boss tells you to make your system you are developing “safe”. This talk:

- Is about what can be done on the *C++ language level* to create a “safe” system.
- Is based on the requirements of ISO 26262 (the automotive functional safety standard). Other safety standards might have different requirements.
- Is *not* about the remaining 99% of functional safety, such as building a safety case, the safety analysis, (semi)-formal validation, simulation, testing, redundancy etc. etc.



## Outside of certification context

The information in this talk is also useful for engineers not working in a regulated environment.





**Safe:** Absence of unreasonable risk due to hazards caused by malfunctioning behavior of systems.

**Qualified:** A tool, component or library etc. is compliant to relevant safety clauses of a specific safety standard.

**Certified:** An independent 3<sup>rd</sup> party safety assessor/agency has analyzed a system (mostly based on reports regarding the system such as safety analysis or code coverage reports) and concluded the product is compliant to a specific safety standard.

Safe,  
qualified,  
certified

# What is needed for safe modern C++ based product

CPP-Summit 2019

## High level requirements

1. Qualified Toolchain
2. Memory static operation
3. Real Time
4. Testability

## This talk will exercise the above requirements on

1. C++ compiler and standard library
2. Standard containers and exceptions
3. Threading and related architecture
4. Failure injection



# What is needed for safe modern C++ based product

CPP-Summit 2019

High level requirements

1. **Qualified Toolchain**
2. Memory static operation
3. Real Time
4. Testability

This talk will exercise the above requirements on

1. **C++ compiler and standard library**
2. Standard containers and exceptions
3. Threading and related architecture
4. Failure injection

## Basic toolkit required for a safe C++ product

- Qualified real time operating system
- Qualified C++ compiler
- Qualified C++ standard library

These 3 components can come in a bundle.

Companies providing such bundles:

Blackberry (QNX), Windriver (VxWorks), Greenhills (Integrity)

Any 3rd party library dependencies also needs to be qualified. This means in practice:

- Avoid large 3rd party libraries (boost, poco, etc)
- 3rd party libraries which are qualified are rare

Reuse efforts of others + minimize dependencies



# What is needed for safe modern C++ based product

CPP-Summit 2019

## High level requirements

1. Qualified Toolchain
2. **Memory static operation**
3. Real Time
4. Testability

## This talk will exercise the above requirements on

1. C++ compiler and standard library
2. **Standard containers** and exceptions
3. Threading and related architecture
4. Failure injection

## Requirements

- Application must be memory static at runtime
- No memory fragmentation should occur
- One should be able to reason about resource usage and resource limits of containers to ensure that those are sufficient during the whole runtime of an application

## Challenges

- Standard C++ containers allocate memory and allocate this memory using different strategies
- Stack memory is very limited, running out of stack is hard to recover from
- Memory pools can fragment, depending on how they are used
- Traceability of resource usage must be given to be able to prove a system does not run out of resources which would introduce a safety hazard

Memory pools and allocators are only one piece of the solution.



# Solution

## Strings (`std::string`)

Compile time fixed size string with stack storage in two flavors, one which silently truncates and one which throws on overflow

- No memory allocation
- Small size makes it well suitable to store on the stack
- Simple to use

## Vectors (`std::vector`)

Runtime fixed size vector which allocates from heap on construction

- Memory for each vector is allocated before steady-time
- Satisfies continuous memory guarantees
- Harder to use as not constructable or copyable during runtime

## Node based containers (`std::map`, `std::set` etc.)

Memory-pool / allocator framework (for example <https://github.com/foonathan/memory>)

- Pools are allocated before runtime
- One pool per type prevents memory fragmentation
- Pools can be arbitrarily granular ensuring resource isolation and makes proving that the application does not run out of resources easier
- Unordered containers do not work with fixed size pools

<https://bduvenhage.me/performance/2019/04/22/size-of-hash-table.html>

Different container types require different solutions

# What is needed for safe modern C++ based product

CPP-Summit 2019

## High level requirements

1. Qualified Toolchain
- 2. Memory static operation**
3. Real Time
4. Testability

## This talk will exercise the above requirements on

1. C++ compiler and standard library
2. Standard containers and **exceptions**
3. Threading and related architecture
4. Failure injection



## Requirements

- Application must be memory static at runtime, therefore exceptions need to be memory static
- One should be able to reason about resource usage and resource limits of exception management to ensure that those are sufficient during the whole runtime of an application
- Runtime of routines need to be bounded and well understood

## Challenges (might not be the same for all compilers)

- Throwing an exception allocates memory on the compiler level
- Standard exceptions allocate memory to store the error strings
- Exception handler lookup might not have deterministic runtime. Lookup time could be dependent on inheritance structure.
- Exception handling adds hard to track and hard to analyze execution branches to an application. This complicates runtime and test coverage analysis.

Exceptions allocate memory!

# Memory static and deterministic exceptions

## Modify how compiler allocates memory for exception

Example on how to do this for GCC: ApexAI Static Exception

```
extern "C" void * __cxa_allocate_exception(size_t
thrown_size);
extern "C" void __cxa_free_exception(void *thrown_object);
extern "C" __cxxabiv1::__cxa_dependent_exception *
__cxa_allocate_dependent_exception();
extern "C" void __cxa_free_dependent_exception
(__cxxabiv1::__cxa_dependent_exception *
dependent_exception);
```

Different compilers  
required different  
solutions

## Modify the standard C++ library to use a string storage which does not allocate

We're still working on this

## Make exception handler selection real time (e.g avoid dynamic cast).

We're still working on this



## AV Rule 208

**Rationale:** Tool support is not adequate at this time.

- Exception handling creates additional branches which are hard to cover with tests and 100% branch coverage is mandatory for the highest safety levels

# Tool support for exceptions is still a challenge

© 2019 Apex.AI, Inc.



# What is needed for safe modern C++ based product

CPP-Summit 2019

## High level requirements

1. Qualified Toolchain
2. Memory static operation
- 3. Real Time**
4. Testability

## This talk will exercise the above requirements on

1. C++ compiler and standard library
2. Standard containers and exceptions
- 3. Threading and related architecture**
4. Failure injection



## Requirements

- Tasks need to have a deterministic runtime
- Execution of a task needs to be interruptible by a higher priority task
- Lower priority task must not block higher priority ones

## Challenges

- Standard C++ threading library only provides very basic control over thread priorities, priority inheritance, CPU pinning etc.
- Execution in C++ is not preemptible (yet), making it hard to write any kind of real-time capable executor
- It is very difficult to ensure multithreaded code is correct because code execution is interleaved in a non-deterministic way
- If a data race occurs, the behavior of the program is undefined

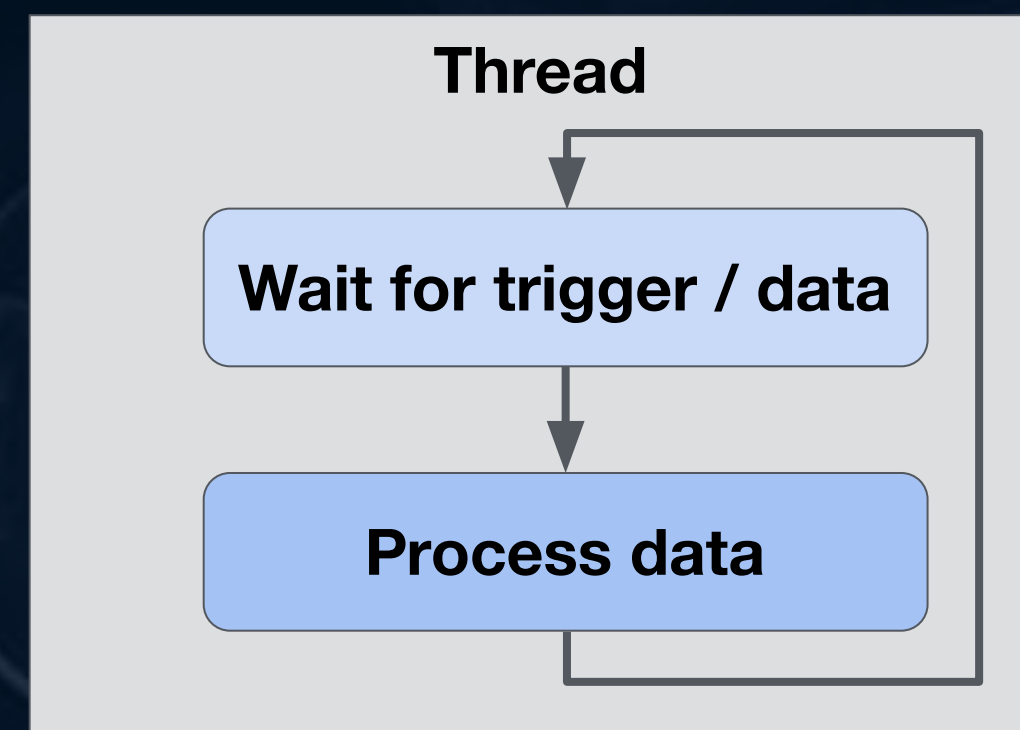
## Custom standard threading library

A custom modern C++ threading library abstracts away the POSIX interface, which makes creating multithreaded application easier and hides implementation details

- Threads are created during initialization, but put on hold until started for runtime
- Thread priority, CPU pinning, etc. are configurable
- Mutexes support priority inheritance

## Rely on the OS Scheduler

- Well understood
- Good tool support
- Disadvantage: More context switches than with an executor based architecture



## Use advanced tools to ensure correctness

- Thread Sanitizer
- Clang Thread Safety Analysis
- Hellgrind

Library support,  
Software architecture  
and tools need to be  
in place



# What is needed for safe modern C++ based product

CPP-Summit 2019

## High level requirements

1. Qualified Toolchain
2. Memory static operation
3. Real Time
- 4. Testability**

This talk will exercise the above requirements on

1. C++ compiler and standard library
2. Standard containers and exceptions
3. Threading and related architecture
- 4. Failure injection**

# Requirements and Challenges

## Requirements

- Code must have very high test coverage
- Coverage gaps must be manually inspected for safety

## Challenges

- Error handling code must be executed in test cases which can be hard to achieve

```
class Thread {
Thread() {
    if(pthread_create(&t, nullptr, f, nullptr)) {
        // How do we test the error handling in this condition?
        throw std::runtime_error("...");
    }
}
pthread_t t;
};
```



# Classical Approaches to Failure Injection

- Macros: Pollute production code, difficult to scale

```
APEX_SKIP_IF(FI_MODE_0, (mtx_ptr = apex_calloc(1U, sizeof(*mtx_ptr))));
if (mtx_ptr != NULL) {
    int32_t result = -1;
    APEX_DO_IF(FI_MODE_1, (result = -1));
    APEX_SKIP_IF(
        FI_MODE_1,
        (result = pthread_mutexattr_init(&mtx_ptr->m_attr)));
    if (0 == result) {
        APEX_DO_IF(FI_MODE_2, (result = -1));
        APEX_SKIP_IF(
            FI_MODE_2,
            (result = pthread_mutexattr_setprotocol(&mtx_ptr->m_attr, PTHREAD_PRIO_INHERIT)));
        if (0 == result) {
            APEX_DO_IF(FI_MODE_3, (result = -1));
            APEX_SKIP_IF(
                FI_MODE_3,
                (result = pthread_mutex_init(&mtx_ptr->m_mutex, &mtx_ptr->m_attr)));
        }
    }
}
```

- Replace shared library with mock library: No pollution of production code but hard to maintain and not always applicable

C++ provides additional tools in your failure injection toolkit



Class wrapper for functions which should inject failures. Avoid logic in this wrapper as this logic would need tests again.

Mock class which triggers the desired failure. Failure behaviour can also be made configurable to help covering all possible failure modes.

Use a template parameter to inject the desired wrapper.

Using a typedef to hide failure injection from the user.

In test code use the mock wrapper as template argument.

```
struct PThreadWrapper {
    static int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                             void *(*start_routine) (void *), void *arg) {
        return pthread_create(thread, attr, start_routine, arg);
    }
};

struct PThreadWrapperMock {
    static int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                             void *(*start_routine) (void *), void *arg) {
        return -1;
    }
};

namespace detail {
    template<class ImplT>
    class Thread {
    public:
        Thread() {
            if(ImplT::pthread_create(&t, nullptr, f, nullptr)) {
                throw std::runtime_error("...");
            }
        }
        pthread_t t;
    };
}

using Thread = detail::Thread<PThreadWrapper>;

// In production code do:
Thread thread;

// To test do:
detail::Thread<PThreadWrapperMock> test_thread;
```



# C++ Approaches to Failure Injection: Polymorphism

CPP-Summit 2019

Polymorphic interface for the function which should inject failures.

Implementation which does the real function call.

Mock implementation which injects failures.

A factory which selects either the real or the mock implementation based on some configuration.

Implementation uses factory to get interface.

In test code set the configuration to let the factory generate the desired failure injecting mock.

```
struct PThreadBase {
    virtual int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                              void *(*start_routine) (void *), void *arg) const = 0;
};

struct PThreadWrapper : PThreadBase {
    int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                      void *(*start_routine) (void *), void *arg) const override {
        return pthread_create(thread, attr, start_routine, arg);
    }
};


struct PThreadWrapperMock : PThreadBase {
    int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                      void *(*start_routine) (void *), void *arg) const override {
        return -1;
    }
};

bool g_in_test = false;
const PThreadBase & pthread_factory() {
    if(g_in_test) {
        static PThreadWrapperMock wrapper;
        return wrapper;
    } else {
        static PThreadWrapper wrapper;
        return wrapper;
    }
}

class Thread {
public:
    Thread() {
        if(pthread_factory().pthread_create(&t, nullptr, f, nullptr)) {
            throw std::runtime_error("...");
        }
    }
    pthread_t t;
};

// In production code do:
Thread thread;

// To test do:
void test() {
    g_in_test = true;
    Thread test_thread;
}
```

The background of the slide is a dark blue gradient. On the left side, there is a faint, semi-transparent image of a car, possibly a truck or a large SUV, viewed from the front. A prominent white diagonal line runs from the top right corner towards the bottom left, separating the dark blue area from a white area on the right.

# Thank you!

# Questions?